lcr-modules

Release 1.0

Jun 09, 2020

For Users

CHAPTER 1

Getting Started With lcr-modules

- **Users:** Check out this *Getting Started* guide.
- **Contributors:** Check out this *Getting Started* guide.

CHAPTER 2

Motivation

This project aims to become a collection of standard analytical modules for genomic and transcriptomic data. Too often do we copy-paste from each other's pipelines, which has several pitfalls:

```
* Too much time spent on routine analyses                    * Increased risk for hidden␣
↪logical bugs
* Duplicated effort within and between labs                  * No consistently used pipelining␣
↪tool
* Inefficient dissemination of best practices                * Steep learning curve for new␣
↪members
```

Fortunately, all of these problems can be solved with standardized analytical modules, and the benefits are many:

```
* Projects can ramp up faster                                * Consistent intermediate/output␣
↪files
* Streamline efforts between labs                           * More reproducible analyses
* Define analytical best practices                          * Easier-to-write methods
* Consolidate collective expertise                          * Automated logging and "paper␣
↪trail"
* Simplify member onboarding                                * Easier peer review of code

                              * And happier bioinformaticians!
```

CHAPTER 3

What Are Modules?

Each module accomplishes a specific analysis, generally centered around a specific tool (*e.g.* Strelka2, Manta, MutSigCV). Analyses—and by extension, modules—can be organized into different levels. The figure below contains for examples for each level.

- **Level-1 Analyses:** They process raw sequencing data, generally producing BAM/FASTQ files.
- **Level-2 Analyses:** They perform sample-level analyses on level-1 output, such as variant calling and gene expression quantification.
- **Level-3 Analyses:** They aggregate sample-specific level-2 output and perform cohort-wide analyses, such as the identification of sifgnificantly mutated genes.
- **Level-4 Analyses:** They are project-specific and are meant to ask specific questions of the data. These are the analyses you ideally want to spend your time on.

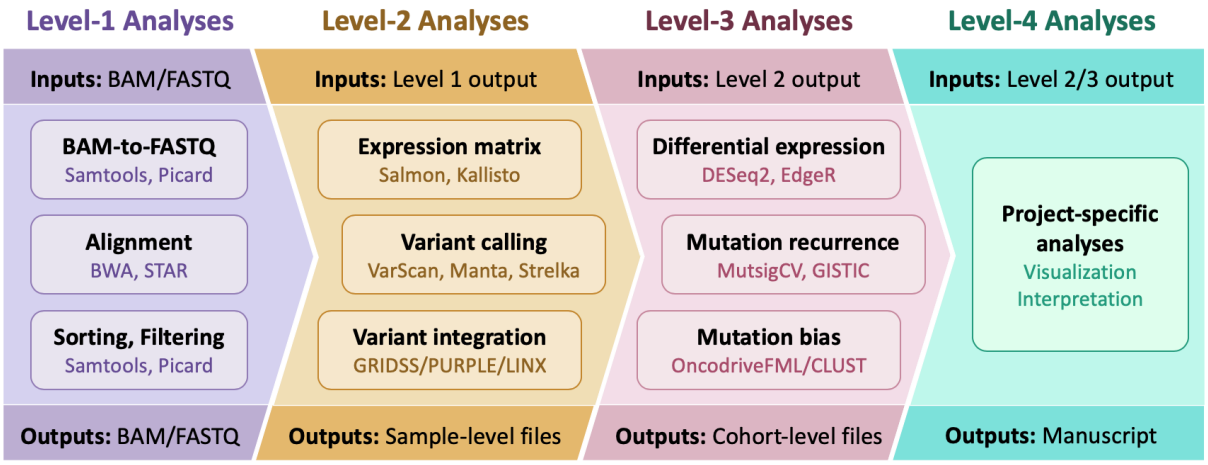

Fig. 1: Module Levels

3.1 Getting Started

Important: Be sure to update any values in angle brackets (`<...>`). File paths can either be absolute or relative to the working directory (usually where you run your `snakemake` command).

1. Install `conda` (Python version 3.6 or later) with Miniconda or Anaconda. Ideally, create a project-specific conda environment. You can install Git using conda if need be.

```
# Optional: Create a project-specific conda environment
# conda create -n <project_name> "python>=3.6"
# conda activate <project_name>

conda install git
```

2. Clone the lcr-modules repository and the lcr-scripts repository.

```
git clone https://github.com/LCR-BCCRC/lcr-modules.git
git clone https://github.com/LCR-BCCRC/lcr-scripts.git
```

3. Install the custom *oncopipe* python package included in the lcr-modules repository, which will also install dependencies such as `snakemake` and `pandas`.

```
cd lcr-modules/
pip install -e oncopipe/
```

4. Test your environment with the Demo Snakefile using the following `snakemake` command. If you want to actually run the test, check out *Running the Demo Project*.

```
cd demo
nice snakemake --dry-run --use-conda all
```

5. Create a *Sample Table* as a tab-delimited file with these *Required Columns*. The following sample table is an example taken from the Demo Project (`demo/data/samples.tsv`).

sample_id	seq_type	patient_id	tissue_status	genome_build
TCRBOA7-N-WEX	capture	TCRBOA7	normal	grch37
TCRBOA7-T-WEX	capture	TCRBOA7	tumour	grch37
TCRBOA7-T-RNA	mrna	TCRBOA7	tumour	grch37

6. Add the *lcr-modules Configuration* to your project configuration YAML file (*e.g.* `config.yaml`). If you have unpaired tumour samples, you will need to add a `pairing_config`. Check out *Handling Unpaired Tumours* section for more information and *Within a Configuration File* for an example of how this is done.

```
# Update `scratch_directory` if you have a space to store large intermediate files
lcr-modules:
   _shared:
      repository: "<path/to/lcr-modules>"
      lcr-scripts: "<path/to/lcr-scripts>"
      root_output_dir: "results/"
      scratch_directory: null
```

7. Load the *Sample Table*, *Specify the Input Samples*, and add the *Reference Files Workflow* by including the following lines in your project snakefile anywhere before the module snakefiles (see next step).

```
import oncopipe as op

SAMPLES = op.load_samples("<path/to/samples.tsv>")
config["lcr-modules"]["_shared"]["samples"] = SAMPLES

subworkflow reference_files:
   workdir:
      "</path/to/reference_directory/>"
   snakefile:
      "<path/to/lcr-modules/workflows/reference_files/1.0/reference_files.smk>"
   configfile:
      "<path/to/lcr-modules/workflows/reference_files/1.0/config/default.yaml>"
```

8. Include and configure the modules you want to run by adding the following lines to your project snakefile.

   Important

   - Any values that need to be updated by the user will be indicated in the default *Module Configuration* by `UPDATE` comments.
   - We recommend following the order shown below: (1) load the default module configuration files; (2) load the project-specific configuration file; and (3) include the module snakefiles.
   - The following example assumes that these values are updated in the project-specific configuration file. For more information, check out *Updating Configuration Values*.

```
# Load the default configuration file for each module
configfile: "<path/to/lcr-modules/modules/manta/2.0/config/default.yaml>"
configfile: "<path/to/lcr-modules/modules/star/1.0/config/default.yaml>"
# ...

# Load your project-specific configuration
configfile: "<config.yaml>"

# Include the snakefile for each module
include: "<path/to/lcr-modules/modules/manta/2.0/manta.smk>"
include: "<path/to/lcr-modules/modules/star/1.0/star.smk>"
# ...
```

9. Launch snakemake by specifying the module target rule(s). See *Snakemake Commands* for suggestions on how to run snakemake.

```
nice snakemake --dry-run --use-conda --cores <cores> _<star>_all _<manta>_all
```

10. If you feel comfortable with the above steps, consider reading through the *Advanced Usage*. For example, you can use *Conditional Module Behaviour* to set different input file paths for different sequencing data types (*e.g.* `genome` and `mrna`).

3.2 Running the Demo Project

Before running the Demo Project, you will need to download the Test Data, which is composed of exome and RNA sequencing data for a follicular lymphoma case. Specifically, the tumour sample has both exome and RNA data, whereas the normal sample only has exome data. We acknowledge the Texas Cancer Research Biobank, Baylor College of Medicine Human Genome Sequencing Center, and the study participants for graciously providing these open-access data. These data are described in more detail in the following paper. **Important:** Before downloading

the data, you must first read and agree to the terms laid out in the Test Data README, which is a requirement for redistributing the data.

> Becnel, L. B. et al. An open access pilot freely sharing cancer genomic data from participants in Texas. Sci. Data 3:160010 doi: 10.1038/sdata.2016.10 (2016).

Once the Test Data is downloaded, you will need to update the placeholders in `demo/data/` with the downloaded files (or symbolic links to the files). At that point, you can technically run the Demo Snakefile by omitting the `--dry-run` option from the command in the *Getting Started* instructions, but you might want to update the value set in `demo/config.yaml` under `scratch_directory` to an space where you can readily store large intermediate files (*e.g.* a directory without snapshots or backups).

3.3 Reference Files Workflow

The `reference_files` workflow serves many purposes. In general, it simplifies the deployment of lcr-modules for any reference genome on any computer system. This approach ensures that the steps required to generate any reference file are tracked in a snakefile, ensuring their reproducibility. This is achieved by:

1. Downloading the genome FASTA files and any additional reference files (*e.g.* Gencode transcript annotations).
2. Converting the additional files to match the same chromosome system as the genome builds (*e.g.* UCSC vs NCBI vs Ensembl)
3. Generate the required reference files from what was downloaded using snakemake rules.

The snippet below needs to be added to your project snakefile before you include any of the individual module snakefiles. It adds the `reference_files` workflow as a sub-workflow in your project snakefile. Essentially, you gain access to the snakemake rules in that workflow, which you can trigger by passing files to the `reference_files()` function. In this case, the module will ask for specific reference files (*e.g.* genome FASTA file, STAR index), and if the file doesn't exist, the `reference_files` sub-workflow will create them. For more details, check out the Snakemake Sub-Workflows documentation.

```
subworkflow reference_files:
      snakefile:
         "<path/to/lcr-modules/workflows/reference_files/1.0/reference_files.smk>"
      configfile:
         "<path/to/lcr-modules/workflows/reference_files/1.0/config/default.yaml>"
      workdir:
         "</path/to/reference_directory/>"
```

The `configfile` field specifies the path to the default `reference_files` configuration YAML file, which contains the information required for the workflow to run. The `genome_builds` section is the most important part of this configuration file. It defines the details for each available genome build, including the download URL. This enables the portability and thus reproducibility of lcr-modules. Each genome build also has a version (*i.e.* `grch37` or `grch38`) and a provider (*e.g.* `ensembl`, `ucsc`). This metadata allows the `reference_files` workflow to automatically convert between the chromosome names of different providers (*e.g.* with and without the `chr` prefix).

The `workdir` field specifies where the reference files will be created. Optionally, you can set this to a location shared between multiple lcr-modules users to avoid duplicating reference data. If you are considering building a shared reference directory, you might want to consider pre-populating it using the `prepare_reference_files.smk` snakefile. This will generate all of the output files that the `reference_files` workflow can produce for every genome build listed in the configuration file mentioned above. If you only need one genome build, you can remove the unnecessary genome builds from the configuration file. A bash script is included in the repository to perform this task:

```
workflows/reference_files/1.0/prepare_reference_files.sh </path/to/reference_
↪directory> <num_cores>
```

(continues on next page)

(continued from previous page)

One caveat with the `reference_files` workflow is that the rules therein don't have any names. This is due to a Snakemake limitation. Rule names had to be omitted because this workflow could be included in more than one module loaded by the user and Snakemake doesn't allow duplicate rule names. As a result, you will be numbered rules (*e.g.* `1`, `2`, etc.) in your snakemake logs, such as the example shown below:

```
Job counts:
     count   jobs
     1       1
     1       2
     1       3
     7       _manta_augment_vcf
     2       _manta_configure
     2       _manta_dispatch
     1       _manta_index_bed
     3       _manta_input_bam
     1       _manta_input_bam_none
     3       _manta_output_bedpe
     7       _manta_output_vcf
     2       _manta_run
     3       _manta_vcf_to_bedpe
     1       _star_input_fastq
     1       _star_output_bam
     1       _star_run
     1       _star_symlink_sorted_bam
     1       _star_symlink_star_bam
     1       all
     40
```

3.4 lcr-modules Configuration

```
# lcr-modules configuration
config["lcr-modules"]
```

One of snakemake's most useful features is the ability to separate the workflow logic in the snakefiles from the tuneable parameters in the configuration files. lcr-modules is configured using *Module Configuration* and *Shared Configuration*. All configuration relating to lcr-modules is stored under the `"lcr-modules"` key in the snakemake `config` variable. This way, there is no risk of messing up any existing configuration created by the user.

Important: For brevity, the configuration under the `"lcr-modules"` key will be referred to as "lcr-modules configuration".

3.4.1 Module Configuration

```
# Module configuration
config["lcr-modules"]["<module_name>"]
```

Each module in the lcr-modules repository comes bundled with a default configuration to get users started. This module configuration is stored in the `config/default.yaml` YAML file in module subdirectory. These YAML files load the module configuration under module name in the lcr-modules configuration. You can see this in the excerpt below, which is taken from the `manta` module:

```
lcr-modules:
   manta:
      inputs:
         # Available wildcards: {seq_type} {genome_build} {sample_id}
         sample_bam: null  # UPDATE
         sample_bai: null  # UPDATE
         augment_manta_vcf: "{SCRIPTSDIR}/augment_manta_vcf/1.0/augment_manta_vcf.py"
      # ...
```

The intent behind these module configuration files is that any field can be (and often should be) updated by the user. In fact, some fields **must** be updated before the module can be run. These are indicated by `UPDATE` comments in the default configuration file. In the above excerpt, the two input files `sample_bam` and `sample_bai` are set to `null` and labelled with `UPDATE` comments, indicating that these must be updated by the user.

Important: Before running any module, you must search for any `UPDATE` comments in the default configuration file. See *Updating Configuration Values* for different approaches on how to override the default configuration for each module.

3.4.2 Shared Configuration

```
# Shared configuration
config["lcr-modules"]["_shared"]
```

One of the components of the *lcr-modules Configuration* is the shared configuration. As the name implies, the purpose of this shared configuration is to provide some common and relevant information to all modules. To avoid clashing with module names, this configuration is stored under the `"_shared"` key. (*What does the underscore prefix mean?*)

Important: The configuration of each module is "merged" with the shared configuration, and when there are conflicts, the module configuration takes precedence. In other words, everything under `"_shared"` is used as default values when configuring each module unless the *Module Configuration* already has a value, which will override the shared value. To demonstrate this point, consider the following shared configuration and the module configuration before merging. Once they have been merged, you can see that `key2` now appears in the module configuration using the value from the shared configuration, whereas the value for `key1` didn't change.

```
# Shared configuration
_shared:
   key1: "a"
   key2: "b"

# Module configuration (before merging)
module_x:
   key1: "x"
   key3: "z"

# Module configuration (after merging)
module_x:
   key1: "x"
   key2: "b"
   key3: "z"
```

This behaviour can be leveraged in a number of ways. For example, by setting `unmatched_normal_id` under the `"_shared"` key, you avoid having to specify that value for every module that performs paired analyses (assuming you have unpaired tumours). That said, if you want to use a different `unmatched_normal_id` for a subset of modules, you can override the shared value. Another useful instance is the sharing of the *Sample Table* between modules. This way, the user doesn't have to repeatedly provide the same sample table to each module. Note that this is

only possible because each module automatically filters the samples based on the sequencing data types (`seq_type`) listed in their *Pairing Configuration*.

3.4.2.1 Common Shared Configuration Fields

- `repository`: This field specifies the file path for the cloned `lcr-modules` repository. This path can be relative to your project snakefile or absolute. **This parameter is required.**
- `lcr-scripts`: This field specifies the file path for the cloned `lcr-scripts` repository. This path can be relative to your project snakefile or absolute. **This parameter is required.**
- `root_output_dir`: This field specifies the directory (*e.g.* `"results/"`) where the modules will produce their output files in their respective subdirectories (*e.g.* `results/star-1.0/`, `results/manta-2.0/`). Technically, this shared parameter is optional, as it will default to `"results/"`.
- `scratch_directory`: This field specifies the directory where large intermediate files can be written without worry of running out of space or clogging snapshots/backups. If set to `null`, the files will be output locally.
- `pairing_config`: This field is optional, but it's useful for specifying the normal samples to use in paired analyses with unpaired tumours. See *Handling Unpaired Tumours* for more details and *Updating Configuration Values* for an example configuration file where this is provided.

Handling Unpaired Tumours

Each module has a pairing configuration (`pairing_config`) in their default configuration file. This `pairing_config` dictates which sequencing data types (`seq_type`) are supported by the module, whether the module runs in paired or unpaired mode for each `seq_type`, and if so, how it performs these analyses for each `seq_type`. This information is ultimately used by the *`oncopipe.generate_runs_for_patient()`* function when producing (or not) tumour-normal pairs. If you want to learn more, check out the *Pairing Configuration* section in the developer documentation.

That said, the user doesn't need to worry about the `pairing_config` unless they have unpaired tumour samples or they wish to configure modules for new sequencing data types (`seq_type`). If they have unpaired tumours, for each `seq_type`, they need to specify which normal sample to use for paired analyses where an unmatched normal sample will be used instead of a matched normal sample. This is done by providing values for `unmatched_normal_id`, as demonstrated in the *Within a Configuration File* section.

If the user wants to configure new sequencing data types, they should check out the *Configuring New Sequencing Data Types* section.

3.4.2.2 Updating Configuration Values

Within a Configuration File

If you followed the *Getting Started* instructions, you should have a section in your project configuration file for `lcr-modules` (with at least the `_shared` sub-section). One approach to updating configuration values is to add to this section. **Important:** One requirement for this to work is that you need to load your project configuration file **after** the default module configuration files. Again, if you followed the *Getting Started* instructions, this should already be the case.

By the way, there is nothing forcing you to store your project-specific configuration in the same file as the lcr-modules configuration. You can easily have a `project.yaml` file loaded near the beginning of your snakefile and a `lcr-modules.yaml` file loaded as described in the *Getting Started* instructions.

One of the main limitations of this approach is that you are restricted to value types that can be encoded in YAML format. For the most part, this means numbers, strings and booleans organized into lists or dictionaries. In other words, this precludes the use of functions as values, such as Input File Functions. If you need to specify functions, you will have to update configuration values *Within the Snakefile*, or use a hybrid approach.

The example YAML file below is taken from the Demo Configuration. You can see that it includes a `pairing_config` under `_shared` to indicate which normal samples to use for unpaired tumours for paired analyses (see *Handling Unpaired Tumours*). It also updates a number of configuration values for the `star` and `manta` modules. All of these fields were labelled with an `UPDATE` comment in the modules' respective default configuration file. The only exception is the `scratch_subdirectories` field for the `star` module, which was updated here to include the `"mark_dups"` subdirectory such that the final BAM files from the module are also stored in the scratch directory.

```
# Taken from lcr-modules/demo/config.yaml
lcr-modules:

   _shared:
      lcr-modules: "../"
      lcr-scripts: "../../lcr-scripts"
      root_output_dir: "results/"
      scratch_directory: "scratch/"
      pairing_config:
         capture:
               unmatched_normal_id: "TCRBOA7-N-WEX"

   star:
      inputs:
         sample_fastq_1: "data/{sample_id}.read1.fastq.gz"
         sample_fastq_2: "data/{sample_id}.read2.fastq.gz"
      reference_params:
         star_overhang: "99"
      scratch_subdirectories: ["star", "sort_bam", "mark_dups"]

   manta:
      inputs:
         sample_bam: "data/{sample_id}.bam"
         sample_bai: "data/{sample_id}.bam.bai"
```

Within the Snakefile

You can always update configurations values within the snakefile **after** the default configuration files have been loaded. The advantage of this approach is that you can update the value to anything that Python allows, including functions. This is incredibly powerful in snakemake thanks to Input File Functions and Parameter Functions. Also, *oncopipe* includes some useful functions that make use of these snakemake features (*e.g. Conditional Module Behaviour*).

The main drawback of this approach is that it can be rather verbose, not very elegant, and as a result, not as readable. For instance, the equivalent of the above YAML file using this approach would look like this:

```
config["lcr-modules"]["_shared"]["pairing_config"] = {
   "capture": {
      "unmatched_normal_id": "TCRBOA7-N-WEX"
   }
}

config["lcr-modules"]["star"]["inputs"]["sample_fastq_1"] = "data/{sample_id}.read1.
↪fastq.gz"
```

(continues on next page)

(continued from previous page)

```
config["lcr-modules"]["star"]["inputs"]["sample_fastq_2"] = "data/{sample_id}.read2.
↪fastq.gz"
config["lcr-modules"]["star"]["reference_params"]["star_overhang"] = "99"
config["lcr-modules"]["star"]["scratch_subdirectories"] = ["star", "sort_bam", "mark_
↪dups"]

config["lcr-modules"]["manta"]["inputs"]["sample_bam"] = "data/{sample_id}.bam"
config["lcr-modules"]["manta"]["inputs"]["sample_bai"] = "data/{sample_id}.bam.bai"
```

Alternatively, some of the redundancy can be avoided by using the Snakemake update_config() Function, as follows. However, this alternative approach isn't much better. It takes up as many (if not more) lines, especially if you format the code to be readable.

```
import snakemake as smk

smk.utils.update_config(config["lcr-modules"]["_shared"], {
   "pairing_config": {
      "capture": {
         "unmatched_normal_id": "TCRBOA7-N-WEX"
      }
   }
})

smk.utils.update_config(config["lcr-modules"]["star"], {
   "inputs": {
      "sample_fastq_1": "data/{sample_id}.read1.fastq.gz",
      "sample_fastq_2": "data/{sample_id}.read2.fastq.gz"
   },
   "reference_params": {"star_overhang": "99"},
   "scratch_subdirectories": ["star", "sort_bam", "mark_dups"]
})

smk.utils.update_config(config["lcr-modules"]["star"], {
   "inputs": {
      "sample_bam": "data/{sample_id}.bam",
      "sample_bai": "data/{sample_id}.bam.bai"
   }
})
```

If you want a simpler syntax, you can consider using the *Convenience Set Functions*. That said, a good compromise might be to store as much of these configuration updates *Within a Configuration File* (*i.e.* anything that isn't a function), and you can update values with functions *Within the Snakefile*.

3.5 Sample Table

sample_id	seq_type	patient_id	tissue_status	genome_build
TCRBOA7-N-WEX	capture	TCRBOA7	normal	grch37
TCRBOA7-T-WEX	capture	TCRBOA7	tumour	grch37
TCRBOA7-T-RNA	mrna	TCRBOA7	tumour	grch37

One of the requirements for using lcr-modules is a sample table. This format was selected for its flexibility. Each sample can be annotated with any amount of metadata, but for the purposes of lcr-modules, there are only a few

Required Columns. These columns allow the modules to understand the relationship between the samples, especially for tumour-normal pairing.

These requirements are encoded in schemas, which are stored and versioned in `schemas/`. These schemas are used in conjunction with Snakemake Validation. If the sample table doesn't confirm to a schemas that is required by a module, the user will given an informative error message. For example, the list of *Required Columns* below is encoded in the `base-1.0.yaml` schema (located in `schemas/base/`). The list of schemas will grow as modules are added with specific metadata requirements (*e.g.* strandedness of an RNA-seq library for expression quantification).

The only format requirement for the sample table is that it is a pandas DataFrame (*i.e.* `pandas.DataFrame`). Hence, the format of the file on disk doesn't matter. If you wish to use the *`oncopipe.load_samples()`* convenience function, note that it defaults to parsing tab-delimited files, but this can be overriden using the `sep` argument. The advantage of using *`oncopipe.load_samples()`* is that it offers a straightforward method for renaming your columns to comply with the schema(s). See *Loading and Renaming Columns* for examples.

3.5.1 Entity–Relationship Model

Before describing the required columns, it is useful to consider the entities related to each sample, namely `patient`, `biopsy`, `sample`, `library`, `dataset`, and `alignment`. These entities relate to one another in the following ways:

> **Relationships between entities:** Each patient has one or more biopsies (*e.g.* a tumour biopsy and a blood draw; tumour FF and FFPE biopsies). Each biopsy has one or more nucleic acid samples (*e.g.* DNA and RNA). Each sample has one or more sequencing libraries constructed from its nucleic acid samples (*e.g.* whole genome and RNA sequencing libraries for a tumour FF sample). Each sequenced library produces a a set of sequencing reads (*i.e.* a dataset) with one or more alignments (*e.g.* an hg19 and hg38 alignments), although there is generally a "canonical" alignment if more than one exists and thus a one-to-one relationship between datasets and alignments.

While the term "sample" generally refers to nucleic acid samples, lcr-modules uses the term to refer to the units of data that serve as input for the module, *i.e.* usually sequencing data in the form of FASTQ or BAM files. In most projects, there is a simple one-to-one relationship between these files and nucleic acid samples. In more complex projects where nucleic acid samples have more than one data file, the sample IDs will need to incorporate information to prevent duplicates.

3.5.2 Required Columns

Check out the *Loading and Renaming Columns* section if your sample table has some of the required columns under different names. It also features a demonstration of the *`oncopipe.load_samples()`* convenience function you can use to load your TSV/CSV sample table. The *Adding and Transforming Columns* section is useful if you lack some of the required columns or can derive them from existing columns.

3.5.2.1 `seq_type` – Sequencing data type

The most common values for this column are `genome` (whole genome sequencing), `mrna` (RNA sequencing), `capture` (hybridization-capture or exome sequencing), and `mirna` (miRNA sequencing). While `lcr-modules` can handle any value for `seq_type`, the modules are pre-configured for these common values. If you have new `seq_type` values, you can configure them for modules of interest; this is explained in the *Configuring New Sequencing Data Types* section under *Advanced Usage*.

3.5.2.2 `sample_id` – Sample identifiers

Every `seq_type` and `sample_id` pair must be unique. In other words, if a tumour sample was sequenced using different technologies (*e.g.* whole genome and RNA sequencing), you can use the same sample ID since eachdata file will have a different `seq_type` (*e.g.* `genome` and `mrna`, respectively). On the other hand, if you have been naming your samples based on patient ID and you have tumour-normal pairs, you will need to differentiate their sample IDs (*e.g.* with "T" and "N" suffixes). Similarly, if the same tumour sample has both FF and FFPE data files, you will also need to differentiate their sample IDs (*e.g.* with "FF" and "FFPE" suffixes).

3.5.2.3 `tissue_status` – Tumour or normal

This column classifies the samples as either `tumour` (or `tumor`) and `normal`. This information is required for tumour-normal paired analyses such as somatic variant calling. If you lack a matched normal samples, most modules support being run with an unmatched normal sample with the obvious caveats that the results will not be as clean. Check out *Handling Unpaired Tumours* for more information on how to achieve this.

3.5.2.4 `patient_id` – Patient identifiers

This column groups samples that originate from the same patient, *i.e.* that share the same underlying germline sequence. This information is primarily used in conjunction with the `tissue_status` column to generate all possible tumour-normal pairs from the list of samples.

3.5.2.5 `genome_build` – Reference genome build

This column is only required if you have alignment (*i.e.* samples) using different genome builds. Otherwise, `lcr-modules` will assume that the single set of reference data (*e.g.* `lcr-modules/references/hg38.yaml`) that you load is the one to use.

3.5.3 Loading and Renaming Columns

For your convenience, the *`oncopipe.load_samples()`* function is provided to easily load your samples as a pandas DataFrame. By default, the function assumes tab-delimited files, but you can change this using the `sep` argument. The function can also convert some of your columns to lowercase using the `to_lowercase` argument, which is useful to comply with some of the schemas. By default, it converts the `tissue_status` column to lowercase. It thus becomes trivial to load a sample table.

```
import oncopipe as op
SAMPLES = op.load_samples("samples.tsv")
```

If your sample table uses different column names than those listed in *Required Columns*, you can use the *`oncopipe.load_samples()`* function to rename your columns. For example, let's say you already have a sample table, but the sample ID and patient ID columns are named `sample` and `patient` rather than `sample_id` and `patient_id`. You can easily achieve this as follows:

```
import oncopipe as op
SAMPLES = op.load_samples("samples.tsv", sample_id = "sample", patient_id = "patient")
```

Alternatively, if the column names in your sample table differ systematically from the expected column names, you can rename them by passing a function to the `renamer` argument. You can also pass an anonymous `lambda` function. For instance, if you use two-letter prefixes with a period delimiter to indicate which entity a column describes (*e.g.* `pt.` for patient-related columns, `lb.` for library-related columns, etc.), you can remove the prefix from all columns using a regular expression with the following code:

```
import re
import oncopipe as op
remove_prefix = lambda x: re.sub(r"[a-z]{2}\.", "", x)
SAMPLES = load_samples("samples.tsv", renamer=remove_prefix)
```

3.5.4 Adding and Transforming Columns

If your sample table is missing a required column that has the same value for every sample (*e.g.* `genome_build`), you can easily add the missing column in your snakefile using standard pandas syntax as follows:

```
import oncopipe as op
SAMPLES = op.load_samples("samples.tsv")
SAMPLES["genome_build"] = "hg38"
```

On the other hand, if your sample table is missing a required column that has different values for different samples, you can handle this one of two ways. If you can derive the missing column from existing columns, you can use standard pandas syntax to fill in the missing column. Otherwise, you can always resort to manually adding the missing column in the sample table on disk. The example below shows how the pandas syntax can be used to derive a `tissue_status` column by checking whether the `sample_id` column ends with the letter "T".

```
import oncopipe as op
SAMPLES = op.load_samples("samples.tsv")
SAMPLES["tissue_status"] = SAMPLES["sample_id"].str.endswith("T").map({True: "Tumour",
↪ False: "Normal"})
```

A similar approach can be taken if you have the columns, but they are formatted differently. For instance, if you encoded your sequencing data types as `WGS` and `Exome` instead of `genome` and `capture`, respectively, you can use the `map()` method to switch to the expected values, as follows:

```
import oncopipe as op
SAMPLES = op.load_samples("samples.tsv")
SAMPLES["seq_type"] = SAMPLES["seq_type"].map({"WGS": "genome", "Exome": "capture"})
```

3.5.5 Specify the Input Samples

Once you have a sample table, you need to inform the modules which samples they need to run on. Normally, this is accomplished by storing the customized sample table under the `"samples"` key in the *Module Configuration*. Because each module automatically filters for samples whose `seq_type` appear in their respective *Pairing Configuration*, the user doesn't need to worry about pre-filtering the samples. For example, the user doesn't need to filter for RNA-seq samples for the `star` RNA-seq alignment module. That said, if the user had RNA-seq samples they didn't want processed by the `star` module, they can remove the samples in question and set this pre-filtered sample table as the value for the `"samples"` key.

However, since most users will probably want to run all samples through all applicable modules, it is possible to avoid the step of setting the sample table for each module. To skip this step, you can simply set the full sample table under the `"samples"` key in the *Shared Configuration* (`"_shared"`). This is the method used in the *Getting Started* instructions. The *Shared Configuration* section explains why this works.

3.6 Snakemake Commands

Note: Don't forget to update any values in angle brackets (<...>).

3.6.1 Snakemake Profiles

The most convenient way of running snakemake is using snakemake profiles. Each profile contains a YAML file that dictates the default command-line options to use. This way, you don't have to remember all those snakemake options.

3.6.1.1 GSC Snakemake Profiles

Make sure you first install the custom GSC snakemake profiles using these instructions. Then, you can use each profile using these commands.

3.6.2 Explicit Commands

If you prefer to spell out all of the command-line options in your snakemake commands, example commands are included below. These may eventually become out of sync with the above snakemake profiles. Feel free to compare with the list of arguments for local usage or cluster usage.

3.6.2.1 Local Usage

```
# See below for determining <cores>
nice snakemake --printshellcmds --use-conda --cores <cores> <targets>
```

3.6.2.2 Cluster Usage

```
nice snakemake --cluster-sync "srun --partition=all --ntasks=1 --nodes=1 --
↪output=none --error=none --job-name={rule} --cpus-per-task={threads} --mem=
↪{resources.mem_mb}" --max-jobs-per-second=5 --max-status-checks-per-second=10 --
↪local-cores=1 --latency-wait=120 --jobs=1000 --default-resources="mem_mb=2000" --
↪printshellcmds --use-conda <targets>
```

3.6.3 Extra information

3.6.3.1 Determining Value for `--cores`

To determine the number of cores to grant to snakemake, compare the number of installed cores and the current load on the server. These values can either be obtained precisely using the commands below, or they can be estimated by looking at the output of the `htop` command. I generally select a value for `--cores` equal to the number of installed cores minus the server load minus 10-20 to leave some buffer.

```
# Get the number of installed logical cores
nproc
# Get the average server load over the past 5 minutes
cut -d " " -f 2 /proc/loadavg
```

3.6.3.2 Increasing `ulimit`

snakemake tends to spawn A LOT of processes and open A LOT of files depending on the number of running and pending jobs. You may eventually start running into cryptic errors about processors not being able to start or files not being able to be opened. This happens when you run into user limits. You can get around this issue by increasing

the user limits with the `ulimit` command. However, there are hard limits set by administrators that determine the maximum permitted for non-admin users. You can always ask your administrators to increase these hard limits for certain machines to run snakemake.

GSC `ulimit` Setup

GSC users can include the following code in their `.bashrc` file to increase their ulimits based on the server. Notice how the `n104` numbers head node has a much higher hard limit than the other head nodes. This is because it was manually increased when `n104` was the only head node. For this reason, it is recommended that GSC users specically log into `n104` instead of `numbers`, which will assign you to a random head node.

```
# Only change these values for interactive shells
if [[ $- == *i* ]]; then
  if [[ "$HOSTNAME" == "n104" ]]; then
    # Change the max number of processes
    ulimit -u 32768
    # Change the max number of file descriptors
    ulimit -n 100000
  fi
fi
```

3.6.3.3 Creating `nice` Processes

You will notice that the `snakemake` commands below are all prepended with `nice`. Briefly, this has the effect of lowering the priority of your snakemake process. Now, you're probably wondering why would you ever want to do that. Granted, compute resources should be utilized on a first come, first served basis, but in practice, not every user will pay close attention to who is already running jobs on a server.

Ultimately, it doesn't matter whether this act is intentional, an accident, or due to insufficient knowledge of how to manage shared compute resources. If someone launches a job that uses more cores than are available, your snakemake process will be competing for CPU time, and this will make both processes take longer to complete.

In this situation, we should fall back on the motto from the wise Michelle Obama: "When they go low, we go high." In this case, we follow this rule quite literally, because the `nice` command will increase the "niceness" value of your snakemake process, which will cede CPU time to competing processes with lower (usually default) "niceness" values until they're done.

3.6.3.4 Submitting Cluster Jobs Remotely

It is possible to submit jobs to a cluster remotely via SSH. This could be useful in situations where you have quick jobs that you don't want to submit to the cluster, but you also don't want to run locally on the cluster head node. **Important:** This section assumes that you have SSH keys set up, allowing SSH login to the head node without entering a password.

The command below differs from the explicit command above simply by prepending the `srun` command in `--cluster-sync` with `ssh <head_node>`, where `<head_node>` is the cluster head node where you run `srun` normally. You can now increase the value for `--local-cores` (see above for how to determine this value).

```
nice snakemake --local-cores=<cores> --cluster-sync "ssh <head_node> srun --
↪partition=all --ntasks=1 --nodes=1 --output=none --error=none --job-name={rule} --
↪cpus-per-task={threads} --mem={resources.mem_mb}" --max-jobs-per-second=5 --max-
↪status-checks-per-second=10 --latency-wait=120 --jobs=1000 --default-resources="mem_
↪mb=2000" --printshellcmds --use-conda <targets>
```

3.7 Advanced Usage

3.7.1 Directory Placeholders

When specifying any value in the module configuration, you can use the following shorthands as placeholders in the string. They will be replaced with the actual values dynamically. See the *Conditional Module Behaviour*. section below for example usage.

- `{REPODIR}`: The `lcr-modules` repository directory. This corresponds to the `repository` value under `_shared` in the `lcr-modules` configuration.
- `{MODSDIR}`: The current module subdirectory. This corresponds to `{REPODIR}/modules/<name>/<version>`.
- `{SCRIPTSDIR}`: The `lcr-scripts` repository directory. This corresponds to the `lcr-scripts` value under `_shared` in the `lcr-modules` configuration.

3.7.2 Convenience Set Functions

The *Setup Instructions* demonstrate that everything is configured using the same snakemake `config` nested dictionary object, generally under the `"lcr-modules"` key. While transparent, it results in verbose code, such as:

```
config["lcr-modules"]["manta"]["inputs"]["sample_bam"] = "data/{sample_id}.bam"
```

Alternatively, you can use the so-called convenience "set functions" to simplify the code somewhat. In order to use them, you must first enable them. Behind the scenes, the snakemake `config` object is stored internally for easy access.

```
import oncopipe as op

op.enable_set_functions(config)
```

The first set function you can use is *`oncopipe.set_samples()`*, which sets the samples you want to use at the shared or module level. The first argument corresponds to the module name (or `"_shared"`), and all subsequent arguments should be sample tables each formatted as a pandas DataFrame. This function automatically concatenates the data frames that are provided. Here, `SAMPLES` is the complete sample table, whereas `GENOMES` and `CAPTURES` are sample subsets generated from `SAMPLES` using *`oncopipe.filter_samples()`*.

```
import oncopipe as op

op.set_samples("_shared", SAMPLES)
op.set_samples("_shared", GENOMES, CAPTURES)
```

The second function you can use is *`oncopipe.set_input()`*, which sets the given input for a module. Just like `op.set_samples()`, the first argument is the module name, but this function should not be used for `"shared"`. The second argument is the name of the input file as listed in the module's configuration file. Lastly, the third argument is the value you wish to provide for that input file, which generally is a string value containing the available wildcards (once again, as listed in the module's configuration file). That said, you could provide a conditional value as described below in *Conditional Module Behaviour*.

```
import oncopipe as op

op.set_input("manta", "sample_bam", "data/{sample_id}.bam")
```

3.7.3 Conditional Module Behaviour

Sometimes, a parameter or input file depends on some sample attribute. This sample attribute can be stored in the file as a wildcard or in the sample tables as a column. Two functions are available to parameterize virtually anything, namely *oncopipe.switch_on_wildcard()* and `op.switch_on_column()`. These functions are useful for both module users and module developers. Read their documentation for more details, *e.g.* `help(op.switch_on_wildcard)`.

In the example below, I want to override the default Manta configuration and provide the high-sensitivity version for `mrna` and `capture` tumour samples. This piece of code would be added after loading the module configuration but before including the module snakefile.

```
import oncopipe as op

MANTA_CONFIG_OPTIONS = {
    "_default": "{MODSDIR}/etc/manta_config.default.ini",
    "mrna": "{MODSDIR}/etc/manta_config.high_sensitivity.ini",
    "capture": "{MODSDIR}/etc/manta_config.high_sensitivity.ini",
}
MANTA_CONFIG_SWITCH = op.switch_on_wildcard("seq_type", MANTA_CONFIG_OPTIONS)
op.set_input("manta", "manta_config", MANTA_CONFIG_SWITCH)
```

For more information, check out *Conditional Module Behaviour*.

3.7.4 Configuring New Sequencing Data Types

This section is a work in progress, but you should be able to get started by reading the *Pairing Configuration* section in the developer documentation. It's not recommended to add new sequencing data types (`seq_type`) under the `_shared` key because that will trigger all modules to try to run on the new `seq_type`. It's best to configure the `seq_type` on a module-by-module basis.

3.8 Getting Started

Important: Be sure to update any values in angle brackets (`<...>`). If you are new to Git, we recommend reading through this excellent Git Tutorial.

1. First, check the lcr-modules repository to see if the module already exists under the `modules/` directory. Then, check the lcr-modules open issues to see if the module has already been proposed. If so, reach out to the assignee if there is one, or assign yourself if it's unassigned. Otherwise, create a new issue for the module and assign yourself.

   > **Important:** Please make sure you're assigned to a GitHub issue before you start developing the module to avoid duplicating efforts.

2. Research how to best design the module. While there is no expectation that the first version is perfect, it is preferable that an honest attempt is made to collect feedback both from people with experience with the tools involved and from the literature (*e.g.* benchmarking studies).

3. Install `conda` (Python version 3.6 or later) with Miniconda or Anaconda. Ideally, create a conda environment specific for lcr-modules development. Then, install the `cookiecutter` package and, if need be, Git using conda.

   ```
   # Optional: Create a conda environment for lcr-modules development
   # conda create -n lcr-modules "python>=3.6"
   ```

(continues on next page)

(continued from previous page)

```
# conda activate lcr-modules

conda install cookiecutter git
```

4. Clone the lcr-modules repository and the lcr-scripts repository.

```
git clone https://github.com/LCR-BCCRC/lcr-modules.git
git clone https://github.com/LCR-BCCRC/lcr-scripts.git
```

4. Install the custom *oncopipe* python package included in the lcr-modules repository, which will also install dependencies such as `snakemake` and `pandas`.

```
cd lcr-modules/
pip install -e oncopipe/
```

5. Create a new branch from the `master` branch with the format `module/<module_name>/1.0`, where `<module_name>` refers to the core or defining software tool being used in the module. **Important:** Your `<module_name>` should only contain lowercase alphanumerical characters or underscores (*i.e.* no spaces).

```
git checkout master  # Make sure you're on the master branch
git pull --ff-only   # Pull the latest changes from GitHub
git checkout -b "module/<module_name>/1.0"  # Create new branch
git branch  # Confirm you're on the new branch (with the asterisk)
git push -u origin "module/<module_name>/1.0"
```

6. Create a new module based on the *Module Template*. Check out the *Module Template* section for details on the fields requested during module creation.

```
cookiecutter "template/" --output-dir 'modules/'
git add modules/<module_name>/1.0/
git commit -m "Add initial draft of <module_name> version 1.0"
git push origin "module/<module_name>/1.0"
```

7. Update the basic module created from the template, which can be found under `modules/<module_name>/1.0/`. Parts that need to be updated are indicated by `TODO` comments. You can use the New Module Checklist as a guide. These changes should be regularly committed to Git and pushed to GitHub.

```
git add <files>
git commit -m "<commit message>"
git push origin "module/<module_name>/1.0"
```

8. When you are done with your module, commit any remaining changes and merge the *master* branch into your module branch. You shouldn't have any merge conflicts since any new files should be under new versions.

```
git merge master
git push origin "module/<module_name>/1.0"
```

9. Submit a pull request (PR) with your module branch. After pushing to GitHub, you should be able to see a green button to create a new pull request on the lcr-modules repository page pushing the merge commit. If not, you can create one using a link with the following structure:

```
https://github.com/LCR-BCCRC/lcr-modules/compare/master...module/<module_name>/1.0
```

10. Work through the checklist that will appear when you open the PR. Once this checklist is done, you can request someone to review your PR. They can test the module if they have time and/or provide feedback on its design. Finally, once the reviewer(s) are happy, the PR can be merged. Congratulations!

3.9 Module Template

While it is technically possible to create a new module without using the module template, it's not recommended because using the template will ensure that you are following the latest best practices for lcr-modules.

When you run the command listed in the *Getting Started* instructions, you will be asked for the following information:

- `module_name`: This field specifies the short name for the module. The value provided here should match the value used for `<module_name>` in the branch name when following the *Getting Started* instructions.

  Important: This field should only consist of lowercase alphanumerical characters or underscores (*i.e.* no spaces).

- `module_author`: This field specifies the full name of the person who will write the module (presumably the person entering this information).

- `original_author`: This field specifies the full name of the person who originally wrote the Snakefile or script that is being used to inform the module. If the module is being written from scratch, this field can be set to `N/A`.

- `input_file_type` and `output_file_type`: These fields specify the file type of the input and output files, respectively. Generally, these values will be the file extensions (*e.g.* `bam`, `vcf`).

  Important

  - If there is more than one input file type, just list one of them for now. The same applies for the output file type. You'll be able to add more file types in the Snakefile based on the existing structure.
  - Each of these should only consist of lowercase alphanumerical characters or underscores (*i.e.* no spaces).

- `module_run_per`: Possible values are `tumour` and `sample`. This field determines whether the module is intended to be run once per tumour (*e.g.* variant calling modules) or once per sample regardless of tissue status (*e.g.* BAM alignment and processing).

  Additional options will be added later, such as `tumour_cohort` and `sample_cohort` for level-3 modules (see *What Are Modules?* for more details).

- `seq_type.genome`, `seq_type.capture`, and `seq_type.mrna`: Possible values are `paired`, `unpaired`, and `omit`. These fields determine which sequencing data types (`seq_type`) are intended as input for the module and whether each `seq_type` is intended to be run in paired or unpaired mode. The fields correspond to whole genome, hybrid capture-based, and RNA sequencing, respectively. Select `omit` if a `seq_type` is not applicable for the module.

  Important

  - If you selected "sample" for `module_run_per`, then you should use `unpaired` here. If this is a `paired` analysis, you should start over (cancel with Ctrl-C) and select `tumour` for `module_run_per`.
  - If you selected `tumour` for `module_run_per`, you can select `paired` or `unpaired` depending on whether the module is meant to be run on tumour-normal pairs or not.

3.10 Module Description

3.10.1 Module Structure

When you create a new module using the *Getting Started* instructions, you obtain the following files:

```
modules/<module_name>
├── 1.0
│   ├── <module_name>.smk
│   ├── config
│   │   └── default.yaml
│   ├── envs
│   │   └── samtools-1.9.yaml -> ../../../../envs/samtools/samtools-1.9.yaml
│   ├── etc
│   └── schemas
│       └── base-1.0.yaml -> ../../../../schemas/base/base-1.0.yaml
└── CHANGELOG.md
```

- `<module_name>.smk`: This Snakefile contains the rules defining the module. See *Module Snakefile* below for more details.
- `config/default.yaml`: This configuration YAML file contains all of the user-configurable options, such as input files, conda environments, command-line options, cluster parameters, and the pairing configuration (*i.e.* whether/how to run samples as tumour-normal pairs).
- `envs/`: This folder contains symlinks to individual conda environment YAML files from the `envs/` directory, which is found in the root of the repository. These conda environment are generally tool-specific (*e.g.* `samtools`, `star`). Symlinks are used to keep the repository lightweight and promote reuse of conda environments between modules.
- `etc/`: This folder can contain any accessory files required to run the module, such as configuration files (see `manta` module for an example).
- `schemas/`: This folder contains symlinks to individual schema YAML files from the `schemas/` directory in the root of the repository. These schemas determine the required columns in the samples table. Every module should have the `base-1.0.yaml` schema as a minimum requirement. For more information, check out the *Required Sample Metadata* section below. Symlinks are used to keep the repository lightweight and promote reuse of schemas between modules.
- `CHANGELOG.md`: This file contains the release notes for the module. These release notes should list the changes and the rationale for each change.

3.10.2 Module Snakefile

This section will describe the key components of a module snakefile. It uses the `star` module as an example. Note that `CFG` refers to the module-specific configuration. In the case of the `star` module, this would correspond to:

```
config["lcr-modules"]["star"]
```

3.10.2.1 Module Attribution

This section simply lists the individuals who have contributed to the module in one way or another. The `Original Author` refers to the person who wrote the Snakefile or script that was adapted for the module. The `Module Author` refers to the person who either adapted a previously written Snakefile/script or created the module from scratch. Finally, the `Contributors` refers to the list of individuals who have contributed to the module over time, mainly through incremental version updates.

```
##### ATTRIBUTION #####


# Original Author:   Nicole Thomas
```

(continues on next page)

(continued from previous page)

```
# Module Author:      Bruno Grande
# Contributors:       N/A
```

3.10.2.2 Module Setup

There are a few standard components for the module setup and some optional components. Importing standard modules such as `os` (for the `os.remove()` function) is optional. On the other hand, importing the *oncopipe* module is required because it offers a suite of functions that greatly simplify the process of developing modules and facilitate configuration by the user. For brevity, the module is commonly imported with `import oncopipe as op`, which allows the functions to be accessible using the `op` prefix/namespace (*e.g.* `op.as_one_line()`).

The *oncopipe.setup_module()* function call is also required. This function does most of the heavy-lifting behind the scenes to streamline the process of developing modules. The arguments are self-explanatory: `name` is the module name, `version` is the module version, and `subdirectories` is the output subdirectories, which will be numbered automatically by *oncopipe.setup_module()*.

The first and last subdirectories must be `inputs` and `outputs`, and they will be numbered as `00-inputs` and `99-outputs`, respectively. You should name the subdirectories after the tool name or the process, whatever is more evocative and specific (*e.g.* `star` over `align`, or `mark_dups` over `picard`).

Also, it's worth noting that `lcr-modules` use a variant of semantic versioning where major versions represent changes in the number of rules in the module (or changes in the relationship between rules), whereas minor versions reprsent changes in the configuration of the module (*e.g.* command-line parameters).

The `include` statement for the `utils` module is optional. For more information on the `include` statement, you can refer to the Snakemake Includes documentation. The `utils` module contains rules that are generally useful (*e.g.* BAM file sorting, BAM file indexing). It is meant to be included into another module after it has been configured with *oncopipe.setup_module()*. The reason for this is that `utils.smk` makes use of the `CFG` variable to make sure it doesn't interfere with other modules.

Finally, the `localrules` statement is technically optional, but it is recommended to include it in every module. For more information, you can refer to the Snakemake Local Rules documentation. Essentially, when snakemake submits jobs to a cluster, these rules are run locally instead. It is meant for quick rules (*e.g.* symlinking rules) that aren't computationally intensive and could potentially get stuck in the cluster queue for much longer than they take to run.

```
##### SETUP #####


# Import standard modules
import os

# Import package with useful functions for developing analysis modules
import oncopipe as op

# Setup module and store module-specific configuration in `CFG`
# `CFG` is a shortcut to `config["lcr-modules"]["star"]`
CFG = op.setup_module(
    name = "star",
    version = "1.0",
    subdirectories = ["inputs", "star", "sort_bam", "mark_dups", "outputs"],
)

# Include `utils` module
include: "../../utils/1.0/utils.smk"
```

(continues on next page)

(continued from previous page)

```
# Define rules to be run locally when using a compute cluster
localrules:
    _star_input_fastq,
    _star_symlink_in_sort_bam,
    _star_symlink_in_mark_dups,
    _star_output_bam,
    _star_all,
```

3.10.2.3 Module Rules

Input and Output Rules

The input and output rules serve a few purposes. First, they clearly define the entry and exit points of the module, making the module more modular and easier to tie different modules together. Second, they make it clear to anyone exploring the module output directory what the input files were and what the most useful output files (or deliverables) are. Third, by symlinking the most important files in subdirectories with the same name (*i.e.* `99-outputs`), it makes it easier to archive those files (*e.g.* from scratch space to backed-up storage).

You will notice that the *`oncopipe.relative_symlink()`* function is used in the rules below rather than the standard `os.symlink()` function. The different between the two function is explained here: *What is the difference between op.relative_symlink() and os.symlink()?*.

Below is the input and output rules for the `star` module. Because STAR operates on paired FASTQ files, we actually need to symlink two files per sample. While this could have been achieved in two rules, it was simpler to implement as one shared rule. The output file symlinks both the BAM and BAM index (BAI) files at the same time since they need to travel together. Otherwise, I find it useful to output different file types in different subdirectories in `99-outputs`; see the `manta` module for an example, where VCF and BEDPE files are stored separately. In this specific example, the output file rule also deletes an intermediate file. This is being done here to ensure that the downstream file exists before deleting the upstream file.

```
rule _star_input_fastq:
    input:
        fastq_1 = CFG["inputs"]["sample_fastq_1"],
        fastq_2 = CFG["inputs"]["sample_fastq_2"],
    output:
        fastq_1 = CFG["dirs"]["inputs"] + "fastq/{seq_type}--{genome_build}/{sample_
↪id}.R1.fastq.gz",
        fastq_2 = CFG["dirs"]["inputs"] + "fastq/{seq_type}--{genome_build}/{sample_
↪id}.R2.fastq.gz",
    run:
        op.relative_symlink(input.fastq_1, output.fastq_1)
        op.relative_symlink(input.fastq_2, output.fastq_2)

# The other rules, which are normally in between, were omitted

rule _star_output_bam:
   input:
      bam = CFG["dirs"]["mark_dups"] + "{seq_type}--{genome_build}/{sample_id}.sort.
↪mdups.bam",
      bai = CFG["dirs"]["mark_dups"] + "{seq_type}--{genome_build}/{sample_id}.sort.
↪mdups.bam.bai",
      sorted_bam = rules._star_symlink_sorted_bam.input.bam
   output:
      bam = CFG["dirs"]["outputs"] + "bam/{seq_type}--{genome_build}/{sample_id}.bam"
```

(continues on next page)

(continued from previous page)

```
    run:
        op.relative_symlink(input.bam, output.bam)
        op.relative_symlink(input.bai, output.bam + ".bai")
        os.remove(input.sorted_bam)
        shell("touch {input.sorted_bam}.deleted")
```

Target Rules

Generally, the last rule of the module snakefile is the "master target rule". This rule is usually named `_<module_name>_all` (*e.g.* `_star_all`), and expands all of the output files (the files symlinked into `99-outputs`) using either the samples table (`CFG["samples"]`) or the runs table (`CFG["runs"]`) depending on whether the module is run once per sample or once per tumour. The two examples below show a preview of each table and how each can be used in the target rule.

Using the Samples Table

sample_id	seq_type	patient_id	tissue_status	genome_build
TCRBOA7-T-RNA	mrna	TCRBOA7	tumour	grch37

In the example below, since STAR is run on all RNA-seq BAM file, we are using the samples table, which has been automatically filtered for samples whose `seq_type` appears in the module's `pairing_config`. For more information on the `pairing_config`, check out *Pairing Configuration*. Note the use of the `rules` variable that snakemake automatically generates for retrieving the output files from previous rules in the module.

```
rule _star_all:
    input:
        expand(
            rules._star_output_bam.output.bam,
            zip,  # Run expand() with zip(), not product()
            seq_type=CFG["samples"]["seq_type"],
            genome_build=CFG["samples"]["genome_build"],
            sample_id=CFG["samples"]["sample_id"])
```

Using the Runs Table

pair_status	tumour_sample_id	normal_sample_id	tumour_seq_type	normal_seq_type	tumour_patient_id	normal_patient_id	tumour_tissue_status	normal_tissue_status	tumour_genome_build	normal_genome_build
matched	TCRBOA7-T-WEX	TCRBOA7-N-WEX	capture	capture	TCRBOA7	TCRBOA7	tumour	normal	grch37	grch37
no_normal	TCRBOA7-T-RNA		mrna		TCRBOA7		tumour		grch37	

In this second example, taken from the `manta` module, we can see how the runs table (`CFG["runs"]`) is used to define the targets. Because the runs table lists tumour-normal pairs, each column from the samples table is present, but they are prefixed with `tumour_` and `normal_`. The only column that isn't taken from the samples table is `pair_status`, which described the relationship between the tumour-normal pair. Generally, this can be `matched`

if the tumour and normal samples come from the same patient; `unmatched` if the two samples come from different patients; and `no_normal` if there is no normal paired with the tumours.

It's worth noting that the output rule being expanded is `_manta_dispatch` rather than `_manta_output_vcf` and `_manta_output_bedpe`. The reason for this is technical, but briefly, it is because an input file function in the `_manta_dispatch` rule determines which files are converted into BEDPE format.

```
rule _manta_all:
    input:
        expand(
            [
                rules._manta_dispatch.output.dispatched,
            ],
            zip,  # Run expand() with zip(), not product()
            seq_type=CFG["runs"]["tumour_seq_type"],
            genome_build=CFG["runs"]["tumour_genome_build"],
            tumour_id=CFG["runs"]["tumour_sample_id"],
            normal_id=CFG["runs"]["normal_sample_id"],
            pair_status=CFG["runs"]["pair_status"])
```

Other Rules

Every other rule serve to complete the module. These other rules can vary considerably in scope. Therefore, below is a list of guiding principles to follow when designing these rules. These principles simply make it easier for users to achieve what they want. If one of these guidelines gets in the way of designing your module, feel free to employ a different approach, ideally not at the cost of flexibility for the user.

An example rule that follows most of these principles is included below (taken from the `star` module).

1. Each rule should only consist of one command, unless the rule uses standard tools like `gzip` for additional commands. Otherwise, split into multiple rules, optionally connected using `pipe()` or `temp()` to avoid intermediate files.

   This guideline ensures that rules are modular and can easily be rearranged by the user. It also enables tool-specific conda environments (*e.g.* `samtools`, `star`) to be used, which is not possible is more than one tool is used in a rule.

2. For `input` files, use `rules` references to previous output (or input) files wherever possible.

   These `rules` references minimizes the risk that two files get out of sync, *e.g.* if you update an upstream output file and forget to update every downstream occurrence of that file.

3. Reference data should be provided as input files and ideally have rules in the `reference_files` workflow so they can be generated automatically. If a reference file has parameters, these can be exposed to the user under the `reference_params` section in the module configuration.

   Having reference data as input files ensures that rules are re-run if the reference data is updated. For more information on the `reference_files` workflow, check out the *Reference Files Workflow* section.

4. The `output` (and `input`) files should use values in the `CFG["dirs"]`, which correspond to the subdirectory names provided to `setup_module()`.

   This allows the user to easily adjust the output directory for the entire module.

5. Avoid using non-standard wildcards. The standard wildcards for sample-based modules are: `seq_type`, `genome_build`, and `sample_id`. The standard wildcards for tumour-based modules are: `seq_type`, `genome_build`, `sample_id`, `tumour_id`, and `normal_id`.

Adding new wildcards makes it hard to connect different modules together. For example, if module A adds an `ffpe_status` wildcard and module B depends on module A, module B will have to include `ffpe_status` as a wildcard, even though it's not relevant to module B. You can thus see how this would result in the steady accumulation of wildcards. To change the behaviour of a module/rule based on sample metadata, see the *Conditional Module Behaviour* section below.

6. For `log` files, use the corresponding subdirectory names in `CFG["logs"]`.

   The directories in `CFG["logs"]` are automatically timestamped, which allows the log files from each run to be stored separately for posterity.

7. Store `stdout` and `stderr` in separate `log` files, unless the tool outputs to `stdout`, in which case only `stderr` needs to be stored.

   Storing `stdout` and `stderr` in separate files makes it easier to know what output came from where, and it prevent potential issues with truncated log files.

8. Create an `opts` entry under `param` for all command-line options that are not linked to a `{...}` value, which are configured in the `default.yaml` file.

   As you can see in the example below, every option under `shell` is associated with a value taken from the rule (*e.g.* `--genomeDir {input.index}`), whereas it completely lacks "standalone options" (*e.g.* `--runMode alignReads`). This guideline is to allow the user to have absolute control over the parameterization of the command-line tool.

9. Re-use (or provide) tool-specific conda environments for each rule needing one, which are configured in the `default.yaml` file. This can be skipped if the rule only uses standard UNIX tools (*e.g.* `gzip`, `awk`) or if it uses the `run` directive (instead of the `shell` directive).

   Conda environments simplify software installation for a module and ensure reproducibility by specifying tool versions. Even if a rule only uses standard UNIX tools, it might still be worth using the `coreutils` conda environment to avoid OS variations (*e.g.* GNU vs BSD for `sed`).

10. Add the `threads` and `resources` (`mem_mb`) directives for all non-local rules, which are configured in the `default.yaml` file.

    These directives are essential for running the module on a compute cluster. The values should be as low as possible while ensuring that most jobs are run within a reasonable amount of time (to minimize time spent in the queue).

11. Use the `shell` directive for rules with the `conda` directive. Use the `run` directive instead if more complicated logic is required.

    The *`as_one_line()`* function is meant to be used with the triple-quoted (`"""`) strings for long commands. The benefits of using this function are: (1) spaces are automatically added at the end of each line; (2) double-quotes do not need to be escaped; and (3) cleaner commands that are easier to organize using indentation. For example, any pipes (`|`) or double-ampersands (`&&`) can be indented to indicate the separation between two commands.

```
rule _star_run:
   input:
      fastq_1 = rules._star_input_fastq.output.fastq_1,
      fastq_2 = rules._star_input_fastq.output.fastq_2,
      index = reference_files("genomes/{{genome_build}}/star_index/star-2.7.3a/
↪gencode-{}/overhang-{}".format(
         CFG["reference_params"]["gencode_release"], CFG["reference_params"]["star_
↪overhang"]
      )),
      gtf = reference_files("genomes/{{genome_build}}/annotations/gencode_annotation-
↪{}.gtf".format(
```

(continues on next page)

(continued from previous page)

```
        CFG["reference_params"]["gencode_release"]
    ))
  output:
    bam = CFG["dirs"]["star"] + "{seq_type}--{genome_build}/{sample_id}/Aligned.out.
↪bam"
  log:
    stdout = CFG["logs"]["star"] + "{seq_type}--{genome_build}/{sample_id}/star.
↪stdout.log",
    stderr = CFG["logs"]["star"] + "{seq_type}--{genome_build}/{sample_id}/star.
↪stderr.log"
  params:
    opts = CFG["options"]["star"],
    prefix = CFG["dirs"]["star"] + "{seq_type}--{genome_build}/{sample_id}/",
    star_overhang = CFG["reference_params"]["star_overhang"]
  conda:
    CFG["conda_envs"]["star"]
  threads:
    CFG["threads"]["star"]
  resources:
    mem_mb = CFG["mem_mb"]["star"]
  shell:
    op.as_one_line("""
    STAR {params.opts} --readFilesIn {input.fastq_1} {input.fastq_2} --genomeDir
↪{input.index}
    --outFileNamePrefix {params.prefix} --runThreadN {threads} --sjdbGTFfile {input.
↪gtf}
    --sjdbOverhang {params.star_overhang} > {log.stdout} 2> {log.stderr}
        &&
    rmdir {params.prefix}/_STARtmp
    """)
```

3.10.2.4 Module Cleanup

Every module ends with a clean-up step. At the moment, this mainly consists of outputting the module configuration, including the samples and runs, to disk for future reference. These files are output in a timestampted directory in the `logs/` subdirectory. Additionally, this function will delete the `CFG` variable from the environment to ensure it does not interfere with other modules.

```
# Perform some clean-up tasks, including storing the module-specific
# configuration on disk and deleting the `CFG` variable
op.cleanup_module(CFG)
```

3.10.3 Module Configuration

One of the core principles of `lcr-modules` is configurability, and this is primarily achieved by storing anything that can be adjusted in a configuration file separate from the Snakefile. For most modules, there will be a single configuration file called `default.yaml`. On the other hand, some modules might have multiple configuration files to account for different scenarios. For this reason, there is a `config/` subdirectory for each module where all of these configuration files live.

In theory, configuration YAML files can take on any structure. However, it helps both module users and developers to start with a standard structure. This also facilitates feature development. Below is a description of each section of a typical `default.yaml` file using the `star` module as an example.

3.10.3.1 Configuration Features

Configuration Comments

Make sure that anything that needs to be updated by the user is indicated by an `UPDATE` comment. You can see examples in the excerpts below taken from the `star` default configuration.

Directory Placeholders

Since the module developer won't know where the `lcr-modules` (and `lcr-scripts`, if applicable) repository will be located, one of the features of the `setup_module()` function in *oncopipe* is to replace the following directory placeholders with their actual values. This way, you can specify file paths relative to these directories. See the README for the list of *Directory Placeholders*.

3.10.3.2 Configuring Header

Each module configuration should fall under the `lcr-modules` and `<module_name>` (*e.g.* `star`) keys. The `lcr-modules` top-level configuration key is considered reserved for use by modules in this project and the *oncopipe* package. This ensures that the module configuration is properly siloed and avoids clashes with other configuration set by the user.

```
lcr-modules:
   star:
```

3.10.3.3 Configuring Input and Reference Files

Virtually all modules will have input files, and many will also require reference files. These are defined using the `inputs` and `reference_params` keys, respectively.

The input files will generally be set to `null` and labelled with `UPDATE` comments since they need to be specified by the user. This can be done in the configuration file or in the Snakefile (see the Demo Snakefile for an example). Either way, the available wildcards are usually listed in a comment. If not, you can always look at the wildcards in the output files of the rule using the `inputs` configuration section. In general, these are `{seq_type}`, `{genome_build}`, and `{sample_id}`.

> One advantage of specifying the input files in the Snakefile (as opposed to in the configuration file) is that the user can provide Input File Functions rather than a string.

While conceptually similar to input files, reference files are handled differently in `lcr-modules`. They are generally genome build–specific rather than sample-specific. Accordingly, they need to be generated separately. In the past, this was often done in a time-consuming ad hoc way where the commands used to generate the reference files were often not tracked. A `reference_files` workflow was developed as part of `lcr-modules` to streamline this process and promote reproducibility. Most reference files depend only on the genome build and thus required no intervention from the user since the `genome_build` is a standard wildcard. However, some reference files require additional parameterization (*e.g.* the amount of splice-junction overhang when building a STAR index). These parameters are exposed to the user under the `reference_params` section. Some parameters are so important that they will be commented out with `#!` to require user intervention, such as the `star_overhang` parameter in the example below.

For more information on the approach taken in `reference_files` and its benefits and limitations, check out the section on the *Reference Files Workflow*.

```
inputs:
   # The inputs can be configured here or in the Snakefile
   # Available wildcards: {seq_type} {genome_build} {sample_id}
   sample_fastq_1: "<path/to/sample.R1.fastq.gz>"  # UPDATE
   sample_fastq_2: "<path/to/sample.R2.fastq.gz>"  # UPDATE

reference_params:
   # Ideally, `star_overhang` = max(read_length) - 1
   # STAR indices were precomputed for "74" and "99"
   star_overhang: "99"  # UPDATE
   # The Gencode release to use for the transcript annotation
   gencode_release: "33"
```

3.10.3.4 Configuring Scratch Subdirectories

The `scratch_subdirectories` section provides the user with the ability of storing intermediate files in a scratch directory. Essentially, the listed subdirectories, which must match the names provided to the `subdirectories` argument in *`oncopipe.setup_module()`*, will be made into symlinks to corresponding directories in a scratch space. This scratch space is also specified by the user, generally with the `scratch_directory` key under `_shared`.

Note that if you've already run your Snakefile, the subdirectories will already exist as actual directories and not symlinks. Accordingly, you will have to delete them before adding another entry to `scratch_subdirectories`. Otherwise, you will run into an error.

```
scratch_subdirectories: ["star", "sort_bam"]
```

3.10.3.5 Configuring Options

The `options` section specifies the command-line options for each tool used in the module (where such options exist). Generally, any command-line option not linked to a placeholder (*e.g.* `{input}`, `{output}`, `{params}`) should be listed under the tool's corresponding entry in `options`. This provides the user with ultimate control over how the tool is run without having to deal with the Snakefile.

Even if a tool has no command-line options beyond those already used in the Snakefile, it is useful to include an entry under `options` with an empty string in case options appear in future versions of the tool. For example, if the user wants to use a command-line option available in a later version of a tool, they can update the conda environment (see *Configuring Conda Environments*) and replace the empty string under `options` with the new option, thus avoiding any editing of the underlying Snakefile.

In the example below, you can see that any command-line options associated with a snakemake parameter (*e.g.* `--sjdbOverhang`, `--runThreadN`) or an input/output file (*e.g.* `--readFilesIn`, `--outFileNamePrefix`) are not included here. Instead, they reside in the `star` snakefile.

```
options:
   star:
      --runMode alignReads
      --twopassMode Basic
      --genomeLoad NoSharedMemory
      --readFilesCommand zcat
      --outSAMtype BAM Unsorted
      --outSAMattrIHstart 0
      --chimOutType WithinBAM SoftClip
      --chimSegmentMin 20
```

(continues on next page)

(continued from previous page)

```
   utils_bam_sort: ""
   utils_bam_markdups: ""
   utils_bam_index: "-b"
```

You will also notice the various `utils_bam_*` fields. These correspond to rules in the `utils` module. For example, the following `utils` rule can index a BAM file, and you can see how it has an `opts` parameter that looks up the `utils_bam_index` field under `options`. If it doesn't find a value, it defaults to `"-b"`. In this case, the module developer exposed these fields to the user by including them in the `star` default configuration. In this case, the same default values as in the `utils` module were used, but that might not always be the case depending on the module.

```
# _utils_bam_index: Index a BAM file
rule:
   input:
      bam = CFG["dirs"]["_parent"] + "{prefix}/{suffix}.bam"
   output:
      bam = CFG["dirs"]["_parent"] + "{prefix}/{suffix}.bam.bai"
   # ...
   params:
      opts = CFG["options"].get("utils_bam_index", "-b"),
      prefix = CFG["dirs"]["_parent"] + "{prefix}/{suffix}"
   # ...
   shell:
      op.as_one_line("""
      samtools index {params.opts} -@ {threads}
      {input.bam} > {log.stdout} 2> {log.stderr}
      """)
```

3.10.3.6 Configuring Conda Environments

The conda environments that power each module are listed under `conda_envs`. These allow for specific versions of tools to be automatically installed, which facilitates reproducibility. Each module will specify a set of default versions of each tool. The user can update this conda environments (*e.g.* to use a more recent version), but this might break the module if there are backwards-incompatible changes to the tool's command-line interface.

Each conda environment should ideally be tool-specific because that promotes re-use of environments between modules. Otherwise, commonly used tools such as `samtools` would be included in multiple module-specific environments. This also allows for easier tracking of the tool versions in the file names. This can only be achieved if each module rule is indeed only using one tool, which should be the case.

Note that Snakemake expects the paths to be relative to the Snakefile. This is automatically handled by the *`oncopipe.setup_module()`* function, so these paths are expected to be relative to the working directory. In the example below, you can see the `{MODSDIR}` directory placeholder being used such that the paths are portably regardless of where the user stores the `lcr-modules` repository (as long as `repository` is specified under `_shared`). For more information, check out *Directory Placeholders*.

```
conda_envs:
   star: "{MODSDIR}/envs/star-2.7.3a.yaml"
   samtools: "{MODSDIR}/envs/samtools-1.9.yaml"
   sambamba: "{MODSDIR}/envs/sambamba-0.7.1.yaml"
```

3.10.3.7 Configuring Compute Resources

Many users will be launching the modules on a high-performance computing cluster. Hence, all non-local rules should have sensible default values for resources such as CPU (`threads`) and memory (`mem_mb`). These settings should

strike a balance between the time spent waiting in the queue (with higher resource values) and the time spent running (with lower resource values).

- **"threads":** The number of logical cores to allocate. This number is typically passed to a command-line argument such as `--threads` or `--cores`. Make sure to check the tool's actual CPU usage. If it's consistently lower or higher than the specified amount, consider adjusting the value.
- **"mem_mb":** The amount of memory to allocate in megabytes (MB). This number is usually best determined empirically based on actual tool runs. This can be done in a number of ways, including monitoring `top`/`htop` or inspecting "Maximum resident set size" when the command is prepended with `/usr/bin/time -v`.

```
threads:
   star: 12
   utils_bam_sort: 12
   utils_bam_markdups: 12
   utils_bam_index: 6

mem_mb:
   star: 40000
   utils_bam_sort: 12000
   utils_bam_markdups: 8000
   utils_bam_index: 4000
```

3.10.3.8 Pairing Configuration

The `pairing_config` section is where the module is configured to run for each sequencing data type (`seq_type`). Two examples are included below to illustrate how the `pairing_config` is used. Check out the *Pairing Configuration Options* section for more details on each field (*e.g.* `run_paired_tumours`).

In this first example, we continue with the `star` module. Here, the pairing configuration only lists `mrna` (*i.e.* RNA-seq data) as a supported `seq_type`. In the future, additional sequencing data types could be added, such as `mirna` for miRNA sequencing data. For `mrna`, the `star` module is configured to run on all samples in unpaired mode. This is achieved by first disabling paired mode (`run_paired_tumours` as `False`) and then ensuring that any paired tumours are forced to run in unpaired mode (`run_paired_tumours_as_unpaired` as `True`). Setting `run_unpaired_tumours_with` to `"no_normal"` is meant to clarify that the unpaired tumours should be included; otherwise, they would be omitted since the default for `run_unpaired_tumours_with` is `None`.

```
pairing_config:
   mrna:
      run_paired_tumours: False
      run_unpaired_tumours_with: "no_normal"
      run_paired_tumours_as_unpaired: True
```

This second example was taken from the `manta` module. As you can see, the module can handle `genome`, `capture`, and `mrna` data. It treats `genome` and `capture` data the same way, namely by allowing unpaired tumours to be analyzed using unmatched normals (as opposed to a truly unpaired analysis without a normal sample). Also, paired tumours are not unnecessarily run as unpaired. In contrast, `mrna` data is run specifically in an unpaired fashion without a normal sample because tumour RNA-seq alignments generally do not have matched normal RNA-seq data.

```
pairing_config:
   genome:
      run_paired_tumours: True
      run_unpaired_tumours_with: "unmatched_normal"
      run_paired_tumours_as_unpaired: False
   capture:
      run_paired_tumours: True
```

(continues on next page)

(continued from previous page)

```
    run_unpaired_tumours_with: "unmatched_normal"
    run_paired_tumours_as_unpaired: False
mrna:
    run_paired_tumours: False
    run_unpaired_tumours_with: "no_normal"
    run_paired_tumours_as_unpaired: True
```

Pairing Configuration Options

Here's a brief description of each of the options that go into a `pairing_config`. Here, the term "unpaired tumour" refers to tumours that lack a matched normal sample with the same `seq_type`.

- `run_paired_tumours`: Possible values are `True` or `False`. This option determines whether to run paired tumours. Setting this to `False` is useful for naturally unpaired or tumour-only analyses (*e.g.* for RNA-seq), which is normally done while setting `run_paired_tumours_as_unpaired` to True in case there are any paired tumours.
- `run_unpaired_tumours_with`: Possible values are `None`, `"unmatched_normal"`, or `"no_normal"`. This option determines what to pair with unpaired tumours. Specifying `None` means that unpaired tumours will be skipped for the given module. This option cannot be set to `None` if `run_paired_tumours_as_unpaired` is `True`. Specifying `"unmatched_normal"` means that unpaired tumours will be run by being paired with the unmatched normal sample given by `unmatched_normal_id` (see below). Specifying `"no_normal"` means that unpaired tumours will be run without a normal sample. Note that modules need to be specifically configured to be run in paired and/or unpaired mode, since the commands of the underlying tools probably need to be tailored accordingly.
- `unmatched_normal_id`: This option must be set to a sample identifier (`sample_id`) that exists in the *Sample Table*. This option determines which normal sample will be used with unpaired tumours when `run_unpaired_tumours_with` is set to `"unmatched_normal"`. This is only required if you have unpaired tumour samples, even if `run_unpaired_tumours_with` is set to `"unmatched_normal"`.
- `run_paired_tumours_as_unpaired`: Possible values are `True` or `False`. This option determines whether paired tumours should be run as unpaired (*i.e.* separate from their matched normal sample). This is useful for benchmarking purposes or preventing unwanted paired analyses (*e.g.* in RNA-seq analyses intended to be tumour-only).

3.11 Advanced Module Features

3.11.1 Required Sample Metadata

Every module requires the samples table, which contains metadata on the samples being analyzed. The minimum set of columns expected by `lcr-modules` are the `sample_id`, `patient_id`, `seq_type`, and `tissue_status` columns (see *Required Columns* for more info). These requirements are spelled out using schemas in YAML format. The base requirements can be found in `schemas/base/base-1.0.yaml`.

Some modules will need additional metadata (*e.g.* the strandedness of RNA-seq libraries). These extra requirements should also be described in schema files. To promote modularity, each required column should have its own file to promote modularity. An exception can be made for a set of columns should always be present together. The new schemas should be stored in the shared `schemas/` directory and then symlinked into individual modules. Symlinks are used to keep the repository lightweight and promote reuse of schemas between modules.

An example single-column schema file can be found in `schemas/ffpe_status/ffpe_status-1.0.yaml`, where as a multi-column schema file should look like the base schema, *i.e.* `schemas/base/base-1.0.yaml`.

Important: Read the section below on *Conditional Module Behaviour* for an explanation on why you should avoid adding new wildcards beyond the standard ones described in *Other Rules*.

3.11.2 Conditional Module Behaviour

One size doesn't always fit all, so modules sometimes have to tailor their behaviour based on sample attributes. Snakemake offers more than one avenue to implement these conditional behaviours. The simplest approach is to create parallel rules, which will handle samples differently based on the file names, potentially using wildcard constraints. However, this approach has two major issues.

First, the resulting parallel rules are mostly identical except for a few, often minor differences (*e.g.* a single command-line argument). This redundancy violates the DRY Principle, making the module harder to maintain and more vulnerable to bugs. This pitfall can be avoided by merging the two rules and using the *Switch on Wildcard Value* function from `oncopipe` described below.

Second, it requires the module developer to encode the sample attributes in the file names. While this is not a severe limitation on its own, it complicates the task of connecting modules together because the file names in downstream modules will need to include every wildcard from upstream modules. This would not only lead to unsustainably long file names, but the file names of a module shouldn't depend on which modules are upstream to ensure modularity. The accumulation of module-specific wildcards can be avoided using the *Switch on Sample Metadata* function from `oncopipe` described below.

> To give a specific example, let's say the `salmon` module requires the strandedness of the RNA-seq samples, so this information is encoded in the file name, *e.g.* `{sample_id}.{strandedness}.quant`. Once we have quantified gene expression in all RNA-seq samples, we wish to perform cohort-wide correction for library size. Unfortunately, we need to pull the information about strandedness from the sample metadata in order to find the `salmon` output files because it's part of the file names, even though that information isn't relevant to our library size correction module.

Important: The `oncopipe.switch_on_wildcard()` and `oncopipe.switch_on_column()` functions do not currently support *Directory Placeholders*. This issue will track the implementation.

3.11.2.1 Switch on Wildcard Value

You can use the `oncopipe.switch_on_wildcard()` function to dynamically set the value of an input file or parameter for a snakemake rule based on the value of a wildcard. The first argument (`wildcard`) is the name of the wildcard, and the second argument (`options`) is a dictionary mapping possible values for the wildcard to the corresponding values that should be returned.

This dictionary can make use of special keys. The most important one to note is the `"_default"` special key, whose associated value is selected if the wildcard value isn't among the other keys. You should check out `oncopipe.switch_on_wildcard()` to find out about the other special keys. (*What does the underscore prefix mean?*)

By default, the `oncopipe.switch_on_wildcard()` will replace any placeholders (using the same format as the `shell` directive; *e.g.* `{wildcards.seq_type}`) with the actual values. This beheviour can be tweaked with the `format` (default = `True`) and `strict` (default = `False`) optional arguments. See the function docstring for more information on these optional arguments.

An example taken from the `manta` module is included below (only relevant parts are shown). Here, the `_manta_configure` rule needs to use a different configuration file based on the sequencing data type (`seq_type`). Specifically, we wish to provide the high-sensitivity configuration if the `seq_type` is RNA-seq (`mrna`) or capture-based sequencing (`capture`), or the default configuration otherwise. Accordingly, the first argument is `"seq_type"`.

```
rule _manta_configure:
    input:
        config = op.switch_on_wildcard("seq_type", CFG["switches"]["manta_config"])
```

The second argument is a reference to the module configuration (`CFG`), specifically the `switches` section. Since YAML files are parsed as nested dictionaries, it is straightforward to store the mapping between wildcard values and desired return values in the `default.yaml` configuration file. The relevant part from the YAML file is included below.

```
lcr-modules:
  manta:
    switches:
      manta_config:
        _default: "{MODSDIR}/etc/manta_config.default.ini"
        mrna: "{MODSDIR}/etc/manta_config.high_sensitivity.ini"
        capture: "{MODSDIR}/etc/manta_config.high_sensitivity.ini"
```

`CFG["switches"]["manta_config"]` contains the dictionary representation of the `manta_config` section from the YAML file shown above. You can see how the `"_default"` special key is being used here (see *Switch on Wildcard Value* for more info) as well as the `{MODSDIR}` placeholder for the module subdirectory (see *Directory Placeholders* for more info).

```
# This is the dictionary stored in `CFG["switches"]["manta_config"]`
{
    '_default': '{MODSDIR}/etc/manta_config.default.ini',
    'mrna': '{MODSDIR}/etc/manta_config.high_sensitivity.ini',
    'capture': '{MODSDIR}/etc/manta_config.high_sensitivity.ini'
}
```

3.11.2.2 Switch on Sample Metadata

As I mentioned in *Conditional Module Behaviour*, adding wildcards for conditional behaviour in a Snakefile is unsustainable and goes against the core principle of modularity. One workaround is to query the metadata for each sample (or each tumour-normal pair) and to update the tool command accordingly. The approach is similar to a *Switch on Wildcard Value*, but with a few notable differences.

The function to use is `oncopipe.switch_on_column()` where the first argument (`column`) is the column name, the second argument (`samples`) is the samples data frame (typically `CFG["samples"]`), and the third argument (`options`) is a dictionary mapping possible values in the column to the corresponding values that should be returned. This dictionary follows the same structure as the *Switch on Wildcard Value*. An additional albeit optional argument is called `match_on`, which needs to be set to either `"tumour"` (default) or `"normal"` to determine whether the function uses the `wildcards.tumour_id` or `wildcards.normal_id` to look up a sample ID. The function will automatically use `wildcards.seq_type` to also filter on sequencing data type.

> At the moment, this function only works for tumour-based modules (*e.g.* paired variant calling). It should soon be generalized to also work with sample-based modules (*e.g.* STAR alignment). This issue is tracked here.

The code block below shows how we could achieve the same outcome using *`oncopipe.switch_on_column()`* for the example given in *Switch on Wildcard Value*. The only difference other than the function name is the addition of the `samples` argument before providing the same `options` dictionary. By default, the function will use `wildcards.tumour_id` (and `wildcards.seq_type`) to look up the sample in `CFG["samples"]`. In practice, you would simply use *`oncopipe.switch_on_wildcard()`* since `seq_type` is available as a wildcard.

```
rule _manta_configure:
    input:
        config = op.switch_on_column("seq_type", CFG["samples"], CFG["switches"][
↪"manta_config"])
```

3.11.3 Switch on File Contents

The behaviour of some module depends on the contents (or existence) of input or intermediate files. The best way to address this is using Snakemake Checkpoints. They are a bit complicated to implement, but you can look at the `manta` module (version 1.0) for an example. Do note that checkpoints can be slow because the function using the checkpoint is run sequentially for each sample.

3.12 oncopipe package

3.12.1 Module contents

`oncopipe.`**`as_one_line`**(*text*)
Collapses a triple-quoted string to one line.

Line endings do not need to be escaped like in a shell script. Spaces and tabs are stripped from each side of each line to remove the indentation included in triple-quoted strings.

This function is useful for long shell commands in a Snakefile, especially if it contains quotes that would need to be escaped (e.g., in an awk command).

Returns A single line (i.e., without line endings) of text.

Return type str

`oncopipe.`**`check_reference`**(*module_config*, *reference_key=None*)
Ensure that a required reference config (and file) is available.

If there is no 'genome_build' column in *module_samples* and there is only one loaded reference, this function will assume that the loaded reference is the reference to be used.

Parameters

- **`module_config`** (*dict*) – The module-specific configuration, corresponding to *config['lcr-modules']['<module-name>']*.
- **`reference_key`** (*str, optional*) – The key for a required reference file.

Returns

Return type None

`oncopipe.`**`cleanup_module`**(*module_config*)
Save module-specific configuration, sample, and runs to disk.

`oncopipe.`**`combine_lists`**(*dictionary*, *as_dataframe=False*)
Merges lists for matching keys in nested dictionary.

Parameters

- **`dictionary`** (*dict*) – Nested dictionaries where the key names match up.

```
{'genome': {'field1': [1, 2, 3],
            'field2': [4, 5, 6]},
'mrna': {'field1': [11, 12, 13],
         'field2': [14, 15, 16]}}
```

- **as_dataframe** (*boolean, optional*) – Whether the return value is coerced to pandas.DataFrame.

Returns

The type of the return value depends on *as_dataframe*. If *as_dataframe* is False, the output will look like:

```
{'field1': [1, 2, 3, 11, 12, 13],
'field2': [4, 5, 6, 14, 15, 16]}
```

If *as_dataframe* is True, the output will look like:

```
    field1  field2
0        1       4
1        2       5
2        3       6
3       11      14
4       12      15
5       13      16
```

Return type dict or pandas.DataFrame

oncopipe.**create_formatter**(*wildcards*, *input*, *output*, *threads*, *resources*, *strict*)
Create formatter function based on rule variables.

oncopipe.**enable_set_functions**(*config*)
Enable the *set_** oncopipe convenience functions.

Parameters **config** (*dict*) – The Snakemake configuration nested dictionary.

oncopipe.**filter_samples**(*samples*, ***filters*)
Subsets for rows with certain values in the given columns.

Parameters

- **samples** (*pandas.DataFrame*) – The samples.
- ****filters** (*key-value pairs*) – Columns (keys) and the values they need to contain (values). Values can either be an str or a list of str.

Returns A subset of rows from the input data frame.

Return type pandas.DataFrame

oncopipe.**generate_runs**(*samples*, *pairing_config=None*, *subgroups=('seq_type', 'genome_build', 'patient_id', 'tissue_status')*)
Produces a data frame of tumour runs from a data frame of samples.

Here, a 'tumour run' can consist of a tumour-only run or a paired run. In the case of a paired run, it can either be with a matched or unmatched normal sample.

Parameters

- **samples** (*pandas.DataFrame*) – The samples.

- **pairing_config** (*dict, optional*) – Same as *generate_runs_for_patient_wrapper()*. If left unset (or None is provided), this function will fallback on a default value (see *oncopipe.DEFAULT_PAIRING_CONFIG*).
- **subgroups** (*list of str, optional*) – Same as *group_samples()*.

Returns The generated runs with columns matching the keys of the return value for *generate_runs_for_patient()*.

Return type pandas.DataFrame

oncopipe.**generate_runs_for_patient**(*patient_samples*, *run_paired_tumours*, *run_unpaired_tumours_with*, *unmatched_normal=None*, *run_paired_tumours_as_unpaired=False*, ***kwargs*)

Generates a run for every tumour with and/or without a paired normal.

Note that 'unpaired tumours' in the argument names and documentation refers to tumours without a matched normal sample.

Parameters

- **patient_samples** (*dict*) – Lists of sample IDs (str) organized by tissue_status (tumour vs normal) for a given patient. The order of the samples in each list is irrelevant.
- **run_paired_tumours** (*boolean*) – Whether to run paired tumours. Setting this to False is useful for naturally unpaired analyses (e.g., for RNA-seq).
- **run_unpaired_tumours_with** (*{ None, 'no_normal', 'unmatched_normal' }*) – What to pair with unpaired tumours. This cannot be set to None if *run_paired_tumours_as_unpaired* is True. Provide value for *unmatched_normal* argument if this is set to 'unmatched_normal'.
- **unmatched_normal** (*namedtuple, optional*) – The normal sample to be used with unpaired tumours when *run_unpaired_tumours_with* is set to 'unmatched_normal'.
- **run_paired_tumours_as_unpaired** (*boolean, optional*) – Whether paired tumours should also be run as unpaired (i.e., separate from their matched normal sample). This is useful for benchmarking purposes or preventing unwanted paired analyses (e.g., in RNA-seq analyses intended to be tumour-only)
- ****kwargs** (*key-value pairs*) – Any additional unused arguments (e.g, *unmatched_normal_id*).

Returns Lists of sample features prefixed with *tumour_* and *normal_* for all tumours for the given patient. Depending on the argument values, tumour-normal pairs may not be matching, and normal samples may not be included. The 'pair_status' column specifies whether a tumour is paired with a matched normal sample.

Return type dict

oncopipe.**generate_runs_for_patient_wrapper**(*patient_samples*, *pairing_config*)

Runs generate_runs_for_patient based on the current seq_type.

This function is meant as a wrapper for *generate_runs_for_patient()*, whose parameters depend on the sequencing data type (seq_type) of the samples at hand. It assumes that all samples for the given patient share the same seq_type.

Parameters

- **patient_samples** (*dict*) – Same as *generate_runs_for_patient()*.
- **pairing_config** (*nested dict*) – The top level is sequencing data types (seq_type; keys) mapped to dictionaries (values) specifying argument values meant for *generate_runs_for_patient()*. For example:

{'genome': {'run_unpaired_tumours_with': 'unmatched_normal', 'unmatched_normal': Sample(...)},

'mrna': {'run_paired_tumour': False, 'run_unpaired_tumours_with': 'no_normal'}}

Returns Same as *generate_runs_for_patient()*.

Return type dict

oncopipe.**get_from_dict**(*dictionary*, *list_of_keys*)
Access nested index/key in dictionary.

oncopipe.**get_reference**(*module_config*, *reference_key*)

oncopipe.**group_samples**(*samples*, *subgroups*)
Organizes samples into nested dictionary.

Parameters

- **samples** (*pandas.DataFrame*) – The samples.
- **subgroups** (*list of str*) – Columns of *samples* by which to organize the samples. The order determines the nesting order.

Returns The number of levels is determined by the list of subgroups. The number of 'splits' at each level is based on the number of different values in the samples data frame for that column. The 'terminal' values are lists of samples, which are stored as named tuples containing all metadata for that row.

Return type nested dict

oncopipe.**list_files**(*directory*, *file_ext*)
Searches directory for all files with given extension.

The search is performed recursively. The function first tries to use the faster *find* UNIX tool before falling back on a slower Python implementation.

Parameters

- **directory** (*str*) – The directory to search in.
- **file_ext** (*str*) – The file extension (excluding the period).

Returns The list of matching files.

Return type list of str

oncopipe.**load_samples**(*file_path*, *sep='\t'*, *to_lowercase='tissue_status'*, *renamer=None*, ***maps*)
Loads samples metadata with some light processing.

The advantage of using this function over *pandas.read_table()* directly is that this function processes the data frame as follows:

1) Can convert columns to lowercase.
2) Can rename columns using either a renamer function or a set of key-value pairs where the values are the original names and the keys are the desired names.

If a renamer function is provided in addition to a set of key-value pairs, the renamer function will be used first.

Parameters

- **file_path** (*str*) – The path to the tabular file containing the sample metadata (including any required columns).
- **sep** (*str, optional*) – The column separator.

- **to_lowercase** (*list of str, optional*) – The columns to be converted to lowercase.
- **renamer** (*function or dict-like, optional*) – A function that transforms each column name or a dict-like object that maps the original names (keys) to the desired names (values).
- ****maps** (*key-value pairs, optional*) – Pairs that specify the actual names (values) of the expected columns (keys). For example, if you had a 'sample' column while *lcr-modules* expects 'sample_id', you can use:

  load_samples(. . . , sample_id = "sample")

Returns

Return type pandas.DataFrame

oncopipe.**locate_bam**(*bam_directory=None*, *sample_keys=('sample_id'*, *'tumour_id'*, *'normal_id')*, *sample_bams=('sample_bam'*, *'tumour_bam'*, *'normal_bam')*)

Locates BAM file for a given sample ID in a directory.

This function actually configures another function, which is returned to be used by Snakemake.

Parameters

- **bam_directory** (*str, optional*) – The directory containing all BAM files. If None is provided, then the default value of 'data/' will be used.
- **sample_keys** (*list of str, optional*) – The possible wildcards that contain identifiers for samples with BAM files.
- **sample_bams** (*list of str, optional*) – The respective names for the BAM file located for each sample in *sample_keys* in the dictionary returned by the input file function. For example, the BAM file for the sample specified in 'sample_id' wildcard will be stored under the key 'sample_bam' in the returned dictionary.

Returns A Snakemake-compatible input file function taking wildcards as its only argument. This function will return a dictionary of BAM files for any wildcards appearing in *sample_keys* under the corresponding keys specified in *sample_bams*.

Return type function

oncopipe.**relative_symlink**(*src*, *dest*, *overwrite=True*)

Creates a relative symlink from any working directory.

Parameters

- **src** (*str*) – The source file or directory path.
- **dest** (*str*) – The destination file path. This can also be a destination directory, and the destination symlink name will be identical to the source file name (unless directory).
- **overwrite** (*boolean*) – Whether to overwrite the destination file if it exists.

oncopipe.**set_input**(*module*, *name*, *value*)

Use given value for an input file in a module.

Parameters

- **module** (*str*) – The module name.
- **name** (*str*) – The name of input file field. This is usually taken from the module's configuration YAML file.

- **value** (*str or function*) – The value to provide for the named input file. In most cases, this value will be a plain string, but you can also provide an input file function as per the Snakemake documentation, where the function would return strings. In all cases, the strings can make use of the wildcards that are usually listed in the configuration file.

oncopipe.**set_samples**(*module*, **samples*)
Use given samples for a module.

Parameters

- **module** (*str*) – The module name. This can also be `"_shared"` for a value that should be inherited by all modules.
- ***samples** (*list of pandas.DataFrame*) – One or more pandas data frames that will be concatenated before being used by the module. These data frames should contain sample tables as described in the documentation.

oncopipe.**set_value**(*value*, **keys*)
Update lcr-modules configuration using simpler syntax.

This function will automatically create dictionaries if accessing a key that doesn't exist and notify the user.

Parameters

- **value** (*anything*) – The value to be set at the location specified by `*keys`.
- ***keys** (*list of str*) – All subsequent arguments will be collected into a list of strings, which specify the location where to set `value`. You do not need to include the `"lcr-modules"` key; it is assumed that you are accessing keys therein.

oncopipe.**setup_module**(*name*, *version*, *subdirectories*)
Prepares and validates configuration for the given module.

This function performs a number of convenient tasks:

1) It ensures that the *CFG* variable doesn't exist. This is intended as a safeguard since the modules use *CFG* as a convenient shorthand.
2) It ensures that Snakemake meets the required version.
3) It ensures that the required configuration is loaded.
4) It initializes the module configuration with the *_shared* configuration, but recursively overwrites values from the module-specific configuration. In other words, the specific overrides the general.
5) It ensures that the module configuration has the expected fields to avoid errors downstream.
6) It's updates any strings containing placeholders such as *{REPODIR}*, *{MODSDIR}*, and *{SCRIPTSDIR}* with the actual values.
7) It validates the samples table using all of the schema YAML files in the module's *schemas/* folder.
8) It configures, numbers, and creates the output and log subdirectories.
9) It generates a table of runs consisting of tumour- normal pairs in case that's useful.
10) It will automatically filter the samples for those whose *seq_type* appear in *pairing_config*.

Parameters

- **name** (*str*) – The name of the module.
- **version** (*str*) – The semantic version of the module.

- **subdirectories** (*list of str*) – The subdirectories of the module output directory where the results will be produced. They will be numbered incrementally and created on disk. This should include 'inputs' and 'outputs'.

Returns The module-specific configuration, including any shared configuration from *config['lcr-modules']['_shared']*.

Return type dict

oncopipe.**setup_subdirs**(*module_config*, *subdirectories*, *scratch_subdirs=()*)

Numbers and creates module output subdirectories.

Parameters

- **module_config** (*dict*) – The module-specific configuration.
- **subdirectories** (*list of str*) – The names (without numbering) of the output subdirectories.
- **scratch_subdirs** (*list of str, optional*) – A subset of *subdirectories* that should be symlinked into the given scratch directory, specified under:

  config["lcr_modules"]["_shared"]["scratch_directory"]

  This should not include 'inputs' and 'outputs', which only contain symlinks.

Returns The updated module-specific configuration with the paths to the numbered output subdirectories.

Return type dict

oncopipe.**switch_on_column**(*column*, *samples*, *options*, *match_on='tumour'*, *format=True*, *strict=False*)

Pick an option based on the value of a column for a sample.

The function finds the relevant row in *samples* for either the tumour (the default) or normal sample, which is determined by the *match_on* argument. To find the row, the *seq_type* and *tumour_id* (or *normal_id*) wildcards are required.

The following special keys are available:

_default If you provide a value under the key '_default' in *options*, this value will be used if the column value is not among the other keys in *options* (instead of defaulting to "").

_prefix, _suffix If you provide values for the '_prefix' and/or '_suffix' keys in *options*, these values will be prepended and/or appended, respectively, to the selected value (including '_default') as long as the selected value is a string (not a dictionary).

Parameters

- **column** (*str*) – The column name whose value determines the option to pick.
- **samples** (*pandas.DataFrame*) – The samples data frame for the current module.
- **options** (*dict*) – The mapping between the possible values in *column* and the corresponding options to be returned. Special key-value pairs can also be included (see above).
- **match_on** (*{"tumour", "normal"}*) – Whether to match on the *sample_id* column in *samples* using *wildcard.tumour_id* or *wildcard.normal_id*.
- **format** (*boolean*) – Whether to format the option using the rule variables.
- **strict** (*boolean*) – Whether to include the bare wildcards in formatting. For example, if you have a wildcards called 'seq_type', without strict mode, you can access it with *{seq_type}* or *{wildcards.seq_type}*, whereas in strict mode, only the latter option is possible.

This mode is useful if a wildcard has the same name as a rule variable, namely wildcards, input, output, threads, resources.

Returns A Snakemake-compatible input file or parameter function.

Return type function

oncopipe.**switch_on_wildcard**(*wildcard*, *options*, *format=True*, *strict=False*)

Pick an option based on the value of a wildcard for a run.

The following special keys are available:

_default If you provide a value under the key '_default' in *options*, this value will be used if the column value is not among the other keys in *options* (instead of defaulting to "").

_prefix, _suffix If you provide values for the '_prefix' and/or '_suffix' keys in *options*, these values will be prepended and/or appended, respectively, to the selected value (including '_default') as long as the selected value is a string (not a dictionary).

Parameters

- **wildcard** (*str*) – The wildcard name whose value determines the option to pick.
- **options** (*dict*) – The mapping between the possible values in *column* and the corresponding options to be returned. Special key-value pairs can also be included (see above).
- **format** (*boolean*) – Whether to format the option using the rule variables.
- **strict** (*boolean*) – Whether to include the bare wildcards in formatting. For example, if you have a wildcards called 'seq_type', without strict mode, you can access it with *{seq_type}* or *{wildcards.seq_type}*, whereas in strict mode, only the latter option is possible. This mode is useful if a wildcard has the same name as a rule variable, namely wildcards, input, output, threads, resources.

Returns A Snakemake-compatible input file or parameter function.

Return type function

oncopipe.**walk_through_dict**(*dictionary*, *end_fn*, *max_depth=None*, *_trace=None*, *_result=None*, ***kwargs*)

Runs a function at a given level in a nested dictionary.

If *max_depth* is unspecified, *end_fn()* will be run whenever the recursion encounters an object other than a dictionary.

Parameters

- **dictionary** (*foo*) – The dictionary to be recursively walked through.
- **end_fn** (*function*) – THe function to be run once recursion ends, either at *max_depth* or when a non-dictionary is encountered.
- **max_depth** (*int, optional*) – How far deep the recursion is allowed to go. By default, the recursion is allowed to go as deep as possible (i.e., until it encounters something other than a dictionary).
- **_trace** (*tuple, optional*) – List of dictionary keys used internally to track nested position.
- **_result** (*dict*) – Used internally to pass new dictionaries and avoid changing the input dictionary.
- ****kwargs** (*key-value pairs*) – Argument values that are passed to *end_fn()*.

Returns A processed dictionary. The input dictionary remains unchanged.

Return type dict

3.13 How do I handle a conda environment that fails to build?

While conda brings us much closer to computational reproducibility, it isn't perfect. Issues arise when conda packages are removed from Anaconda Cloud or when the dependency resolution algorithm changes. We suggest you try the following steps in order:

1. Remove the build IDs from the conda environment YAML file, although this should already be the case for all environments in `lcr-modules`.
2. Remove the versions for the offending package(s) (*i.e.* the one(s) mentioned in the error message).
3. Remove the offending packages altogether.
4. Remove the dependency packages, leaving only the "target packages". This generally means subsetting to the core conda packages listed in a module's README for the environment in question. While extreme, the hope is that the versions of the dependency packages are not crucial for maintaining scientific reproducibility.
5. Remove the versions for the target packages.
6. If you reach this point, it usually means that a target package is problematic. If possible, replace that package with the same (or similar) version from another Anaconda channel. Ideally, restore the YAML file first and cycle through the previous steps.
7. Install the software tools manually (ideally the versions specified in the YAML file) and ensure they are available in your `PATH` environment variable.

3.14 What does the underscore prefix mean?

The underscore prefix is mainly used to avoid name conflicts. This convention is borrowed from Python. For instance, `collections.namedtuple` has an `_asdict()` method, where the underscore helps prevent clashes with user-defined attributes for the `namedtuple`. For more examples in Python, check out this blog post.

In `lcr-modules`, the underscore prefix is used in a few areas. First, the name of every rule or function defined in a module starts with an underscore followed by the module name (*e.g.* `_manta`). This minimizes the risk for clashing with other rule/function names defined elsewhere by the user, which isn't allowed by Snakemake. Second, the underscore prefix is used for dictionary keys with special behaviour, such as the `"_default"` key in the `op.switch_on_wildcard()` *function*. Third, the shared `lcr-modules` configuration is stored under the `_shared` key, which is done to avoid clashing with a potential module called `shared`.

3.15 What is the difference between `op.relative_symlink()` and `os.symlink()`?

Behind the scenes, `op.relative_symlink()` uses `os.symlink()` while ensuring that the symlinks are relative and correct regardless of the current working directory. This is equivalent to the `-r` option on modern version of the `ln` command-line tool.

3.16 Why am I running into a `NameError: name 'CFG' is not defined` exception?

Each module creates a `CFG` variable as a convenient but temporary pointer to the module configuration (*i.e.* `config["lcr-modules"]["<module_name>"]`). Because each module uses this variable name, the `op.cleanup_module()` function deletes the variable to be safe. Hence, you will run into this `NameError` exception if some code tries to use `CFG` after it's been deleted. If you use `CFG` in the rule directives that are evaluated when the module snakefile is parsed (*e.g.* `input`, `output`, `log`, `params`, etc.), it's not an issue. However, if you use this variable in a function or `run` directive, *i.e.* code that is run after the `op.cleanup_module()` function is run, you will get the error above. You can fix this error by adding this line of code before using the `CFG` variable, which recreates the variable in a local scope:

```
# Replace <module_name> with the actual module name (e.g., `star`)
CFG = config["lcr-modules"]["<module_name>"]
```

3.17 How do I specify the available memory per thread for a command-line tool?

The `mem_mb` resource is meant to represent the total amount of memory used by all threads of a given process. Some tools have command-line arguments allowing the user to specify the amount of memory they can use, such as any Java-based application (*i.e.* using `-Xmx`). In some cases, the tool expects the amount of memory per thread (*e.g.* `samtools sort`), whereas `resources.mem_mb` represents the total amount of memory. Arithmetic expansion in Bash allows you to circumvent this issue as long as you are dealing with integers, which should be the case with `threads` and `mem_mb`. For example, here's how you would divide two integers and print the result: `echo $((12000 / 12))`. We can leverage the same syntax within the `shell` directive of a Snakemake rule. The example below is taken from the `samtools sort` rule in the `utils` module.

```
rule:
    ...
    shell:
        op.as_one_line("""
        samtools sort {params.opts} -@ {threads} -m $({{resources.mem_mb} / {threads}
↪))M
        -T {params.prefix} -o {output.bam} {input.bam} > {log.stdout} 2> {log.stderr}
        """)
```

Python Module Index

o

Index